

Thesis for the degree of Master of Science

A Proxima-based Haskell IDE

Gerbo Engels

October 2008

INF/SCR-07-93



Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Universiteit Utrecht
Utrecht, The Netherlands

Supervisor:
prof. dr. S. Doaitse Swierstra

Daily supervisor:
dr. Martijn M. Schrage

Abstract

Proxima is a generic presentation-oriented editor for structured documents, in which a graphical presentation of the document can be edited directly (*WYSIWYG editing*), resulting in changes in the underlying structure of the document. Proxima understands the document structure and it supports direct edit operations on the document structure as well.

A Proxima instantiation uses a parser to get the document structure; modern compilers have parsers as well, which may be accessible through an API. Using the parser of an external compiler as the Proxima parser saves one from writing a new parser. Furthermore, the parse tree of a compiler can contain extra information that is useful to display in an editor, such as type information of a Haskell compiler in a Haskell source code editor.

This thesis aims to use the parser of an external compiler as the Proxima parser. We identify and solve problems that arise when using the parser of an external compiler as the Proxima parser, and two Haskell editors are developed as Proxima instantiations that use the parser of GHC or EHC respectively.

Contents

1	Introduction	1
1.1	Thesis Overview	2
I	The State of the World	3
2	Existing Haskell Editors and IDE's	5
2.1	Visual Haskell	6
2.2	Eclipse Plug-in	7
2.3	Haskell Editors Written in Haskell	8
2.4	Other Editors or Tools	10
2.5	Conclusion	11
3	Proxima	13
3.1	Overview	13
3.2	A Closer Look	15
4	Haskell Compilers	19
4.1	GHC	19
4.2	EHC	20
5	Motivation	23
5.1	What To Expect in Part II	24
II	Changing the World	25
6	External Parsing	27
6.1	Using Any Haskell Data Type in Proxima	27
6.2	Consequences of Using an External Parser	32
7	Scanning and Layout	33
7.1	Bypassing the Proxima Scanner	33
7.2	Comments	36
7.3	Preprocessing	38
8	Compiler API and Compiler Recommendations	39
8.1	Minimal Requirements of a Compiler API	39
8.2	Recommended API, or Desired Compiler Features	40
9	Implementation	43
9.1	Example	43
9.2	Generic Part of Phi	47
9.3	Single Token Commands	49
9.4	Child Presentation Commands	50
9.5	Miscellaneous Commands	55

9.6	Comments	57
9.7	Phi _{GHC} , Phi _{EHC} and Proxima	58
III	Conclusion & Appendices	63
10	Conclusion	65
10.1	Future Work	66
A	Document Type Grammar	69
B	Installing Phi	71
B.1	Obtaining Proxima	71
B.2	Phi Common	71
B.3	Phi _{GHC}	71
B.4	Phi _{EHC}	72
C	Tutorial: Designing a Phi Editor	73
C.1	Setting Up the Framework	73
C.2	Using the Combinators	77
	Bibliography	80

Chapter 1

Introduction

Interactive development environments (IDE's) offer features that aid the programmer in writing source code. IDE features range from simple support for syntax highlighting of the source code with a naive pattern-matching algorithm, to complete refactoring tools, and even showing type information for the function one is writing. Most editors have implemented these features solely by a pattern match algorithm on the source code, or they analyse the output of a compiler to locate the line and column number where an error appeared. A basic analysis of the source code is quite a cheap operation, but the algorithm doesn't 'understand' what different parts of the code actually mean. For example, in the strongly typed language Haskell [Pey03] the semantic difference between the identifier in a *function* declaration and in a *type* declaration will not be detected. Hence, an editor that supports highlighting Haskell syntax does not know the difference of the use of identifier *foo* in

```
foo :: Int
foo = 3
```

In this example it is most likely that the identifier in the type declaration (*foo :: Int*) will be coloured the same way as in the function declaration (*foo = 3*), although the semantics are different.

IDE's for popular languages such as Java and C++ do support some interaction with an external compiler. By making calls to an external compiler, the IDE can help the programmer in his write-compile-debug cycle. For example, the warnings and errors the compiler returns, which often contain a line and column number, can be interpreted by the IDE to let the programmer quickly navigate to those locations, or the IDE lets the programmer 'step-through' the code using the compiler. This, however, is where most editors stop their interaction with a compiler; response from an external compiler *on the fly* while writing code is rarely seen.

There are systems, such as the Synthesizer Generator [RT84], that do offer real-time response in the editor. The Synthesizer Generator handles computations over the document structure, which are specified with an attribute grammar, but communication with an *external* compiler is hard to accomplish.

The question which arises now is whether there exist editors that have knowledge of the data they work on and that also offer a high degree of integration with an external back-end, preferably with real-time feedback from it.

Proxima [Sch04], developed at Utrecht University, is a (generic, presentation-oriented) *structure* editor. Programming languages—having a well-defined syntax—can be parsed into a meaningful structure, thus Proxima should be able to use such a structure and thus to be aware of the semantics of the data it edits. As a presentation-oriented editor, Proxima can contain editable views of either the source code itself, the parse tree, or any other view as need arises.

The Glasgow Haskell Compiler [GHC] is a mature and the most widely used Haskell compiler and provides an abstract programming interface (API) to access its internal compilation facilities. Proxima, written in Haskell, should be able to use that API and thereby become tightly integrated with an external compiler and have access to, for example, results of type inferencing.

Proxima uses its own parser to obtain the structure of a document, while GHC—or any other compiler—contains a parser too. To avoid the redundancy of parsing the same source code twice, it is desired that Proxima and the external compiler share a parser. This gives us two options: either

Proxima parses the source code and passes the parse tree on to the compiler, or the other way round. The latter is more feasible since it is a tedious job to implement a full parser, whereas a compiler already has a good parser available. Furthermore, if we use the parser of a compiler, *Proxima* will be fully compatible with the compiler, which includes the numerous extensions of GHC, for example.

Unfortunately, the parser for *Proxima* is part of the core of the system, and the parse tree generated by the *Proxima* parser contains extra state information that is not present in the parse tree returned by an external compiler.

This bottleneck forms the main topic for this thesis:

How can we link an external compiler to Proxima, such that the parser of the external compiler can be used as the Proxima parser, and such that Proxima can be used to create an IDE that has a tight integration with the compiler?

To narrow the scope, we focus on GHC as the external compiler, because it offers an API, it is a mature compiler, and it is the most widely used Haskell compiler. Hence, a Haskell IDE as a *Proxima* instantiation that is able to communicate with GHC has a large potential user base and has practical use.

To show that our approach is generic enough, we also implement a *Proxima* instantiation that uses the parsing facilities of EHC [Dij], a Haskell compiler under development at Utrecht University.

1.1 Thesis Overview

The rest of this thesis is organised as follows.

- We give a motivation for this thesis (Chapter 5), but for that we first need to take a look at the current state of editor support for writing Haskell programs (Chapter 2), at *Proxima* (Chapter 3), and at GHC and EHC (Chapter 4). This is Part I.
- In Part II we describe our own contributions. To fit an external compiler into the architecture of *Proxima*, a couple of issues have to be handled, like: what is the impact of bypassing the parsing step of *Proxima* (Section 7.1, Section 6.2) and replacing it with an external parser that is built according to different specifications (Section 6.1). We also give a solution for what to do when the parser throws away data, such as comments (Section 7.2), and make some remarks on preprocessing phases (Section 7.3). This completes Chapter 6 and Chapter 7.

We describe a minimal set of requirements that an API must offer to be able to be used as the back-end for a *Proxima* code editor, as well as a set of desired API features (Chapter 8).

We implemented two editors in *Proxima* (Chapter 9) that use the parser of GHC and EHC (Section 9.7), and abstracted over them to come up with a generic part (Section 9.2). Of course, we came across a number of issues, problems and windfalls, dealing with the compilers or *Proxima*. You find this in Chapter 9.

Notes on Notation

This thesis contains fragments of Haskell code that contain

$\langle \dots \rangle$

in which case some details are omitted or explained later on.

When a qualified name is used, the qualifier is usually meant to be illustrative and to make clear what the origin of a function or type is, such as in

Proxima.parse -- the parser of *Proxima*
ExternalParser.parse -- the parser of an external compiler

The qualifiers may not be defined, but the context makes the intention clear.

Part I

The State of the World

*This Part describes the state of a part of the current world:
what editors are out there for the Haskell hacker,
which compilers do they use,
what is it that is missing in the world of today,
and how does Proxima fit in?*

Chapter 2

Existing Haskell Editors and IDE's

In this chapter, we discuss a number of editors, IDE's, and tools that are used for, or aimed at, writing Haskell programs. We give a short overview of the main features these programs offer and discuss to which extent they can interact with external compilers.

General Editors and IDE's

One can write a program in a simple plain text editor, but most programmers prefer to use a more sophisticated editor or some kind of interactive development environment because these editors and IDE's provide useful features. Some features are basic, such as syntax highlighting, whereas others are more advanced features, such as displaying the type of an element in a pop up, jumping to the location where a method or function is defined, and facilities to manage a project.

Although some IDE's are more advanced than others, nearly all share a common set of features:

- Syntax highlighting
- A virtual file system, which shows the files that belong to a project
- An outline view, such as a tree view that shows the top level blocks (classes, modules) with their elements (methods, variables, etc.) as children
- Quick navigation to function definitions, method declarations, variable instantiations, etc.
- Folding of blocks, which hides parts of the source code to make it more readable
- Integrated build system, where one sets the compilation options and flags once and uses a single button to build the entire project
- A task list/to do list, in which also errors are shown
- Basic refactoring support, such as changing the name of an identifier

These IDE's are language-aware: the language of the source text may influence the behaviour of the features, thus the syntax highlighting for C code is different from Haskell code. There is good support available for the mainstream languages such as C (and C++, C#) and Java, but for less popular languages, such as Haskell there is little editor support, apart from syntax highlighting.

In this chapter we take a look at different Haskell editors that are available or under development.

Preliminary notes on GHC GHC will be mentioned a number of times in the following sections. Section 4.1 explains more about GHC, but for this chapter it is sufficient to know that GHC is a mature Haskell compiler and is considered the flagship of Haskell compilers. Its architecture allows Haskell programs to import GHC itself as a library, thus enabling access to GHC's internal compilation facilities (such as type inference or the parser) through an API.

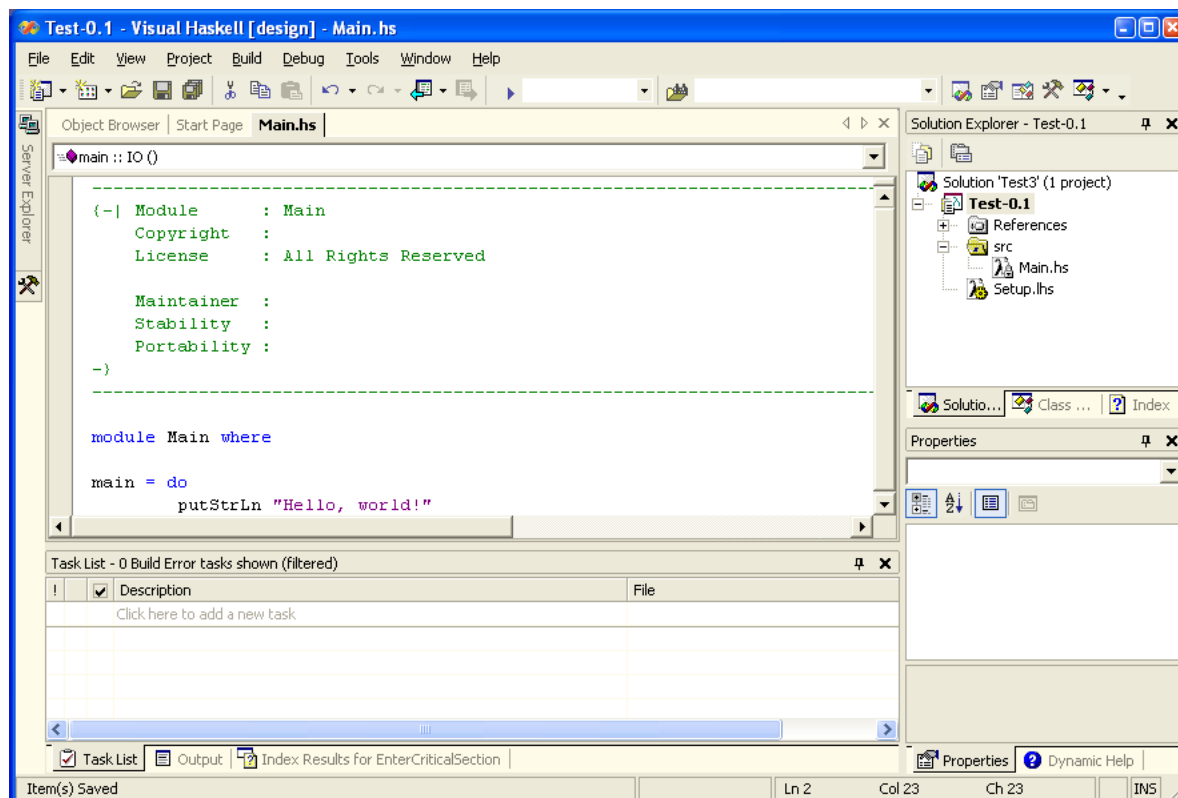


Figure 2.1: Visual Haskell

2.1 Visual Haskell

Visual Haskell [AM05] is a Haskell development environment based on Microsoft's extensible Visual Studio¹ environment. Visual Studio is an advanced and popular IDE that supports various languages and the framework is extensible through the provided COM interface. As a Microsoft product, built on .NET, it is only available on the Windows platform.

Visual Haskell (Figure 2.1) is an extension of Visual Studio. It is an extensive Haskell editor that communicates directly with GHC, which leads to full knowledge of the structure of the source code, including type information. The direct communication helps to automatically check the code for errors as you type, and to show type information for an identifier when the mouse hovers over it. Furthermore, name completion is implemented for identifiers in scope, and one can quickly navigate to a function declaration. The communication with GHC that is needed for these features happens in a background thread, hence it doesn't disturb the interactive feel of the environment. The use of GHC as the back-end for Visual Haskell guarantees that the code is parsed and type checked the same way in the editor as it is during compilation. Therefore it supports not just Haskell98, but also all features and extensions of GHC.

Cabal [Jon05], a system for building and packaging Haskell projects, and Visual Haskell projects are tightly coupled. This means that Visual Haskell projects are Cabal packages, and vice versa. Cabal is also used as the build system. The build errors are communicated back to the editor and put in the task list.

Implementation

The authors of Visual Haskell are (nowadays) part of the Cabal development team. As such, the design of Cabal is heavily influenced by the requirements of Visual Haskell.

To communicate with GHC, the authors had to implement an interface to GHC because that was not available. That API is now also used for GHC's own different front-ends (`ghc --make`, `GHCi`)

¹<http://msdn.microsoft.com/vstudio/>

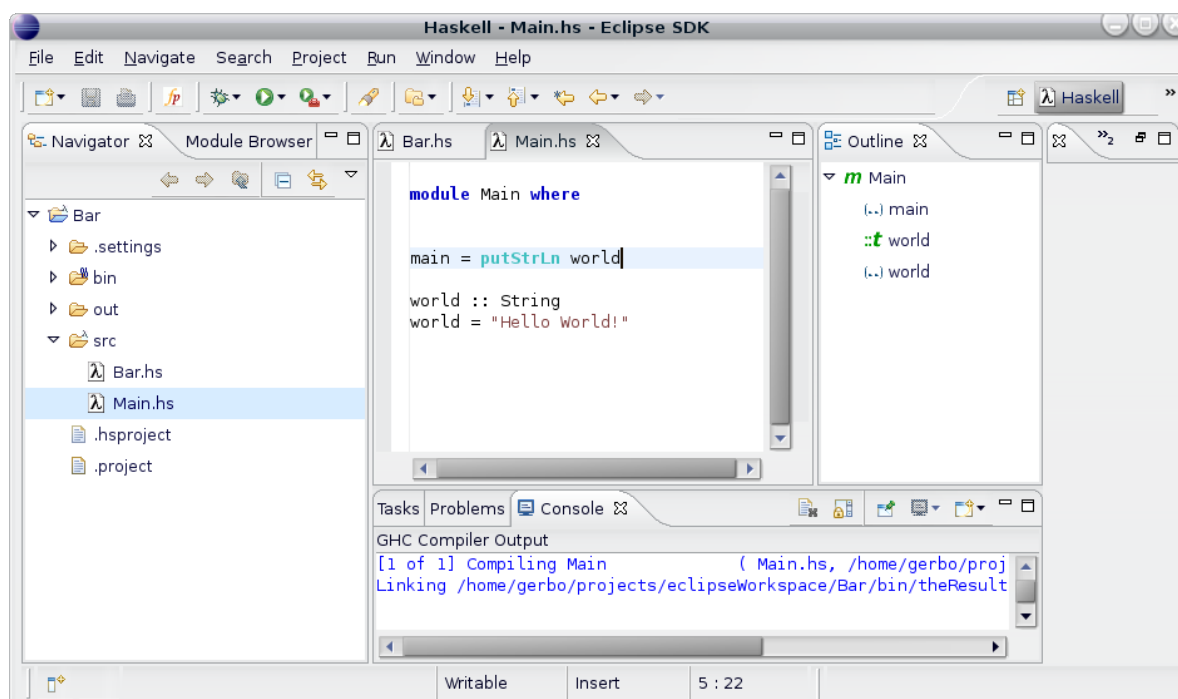


Figure 2.2: Eclipse with the Haskell plug-in

and in other projects that need communication with GHC, thus the interface has facilities designed specifically to be used in a development environment. Most features are implemented in Haskell with the use of this new GHC API.

The next step was to plug everything into Visual Studio. Visual Studio offers a very extensive and complex API to communicate with the environment using COM, as well as a more simple layer that exposes a COM API. Most features are implemented using this simple layer, while some needed to use the more difficult COM API. H/Direct [FLMJ98] provided the bridge between Haskell and COM, but it had to be pushed to the limits. This resulted in code that is difficult to maintain.

Conclusion

It appears that Visual Haskell is a mature tool to write Haskell programs in, and it has a high degree of interaction with GHC. Unfortunately it is not stable enough and occasionally it functions incorrectly.

According to the authors, the implementation seems to be like “90% infrastructure and 10% real features.” Nonetheless, new features can be added with less effort now the basic infrastructure is available.

As an extension to Microsoft’s Visual Studio, Visual Haskell is bound to the Windows platform. It runs only on GHC 6.6 (shipped with the package) because the compiler itself had to be modified to be able to implement all features.

The project is now an open source project, but it is not actively maintained (at the moment of writing, the last update dates back to December 2006).

2.2 Eclipse Plug-in

Another popular extensible development environment is Eclipse². It consists of a base IDE platform written in Java, for which an IDE for a new language can be created by plugging in a set of Java classes.

Leif Frenzel has built a Haskell IDE [Fre07] based on Eclipse, shown in Figure 2.2. The author wants to use Haskell for the implementation, as is the case with the Visual Haskell project, and a lot

²<http://www.eclipse.org/>

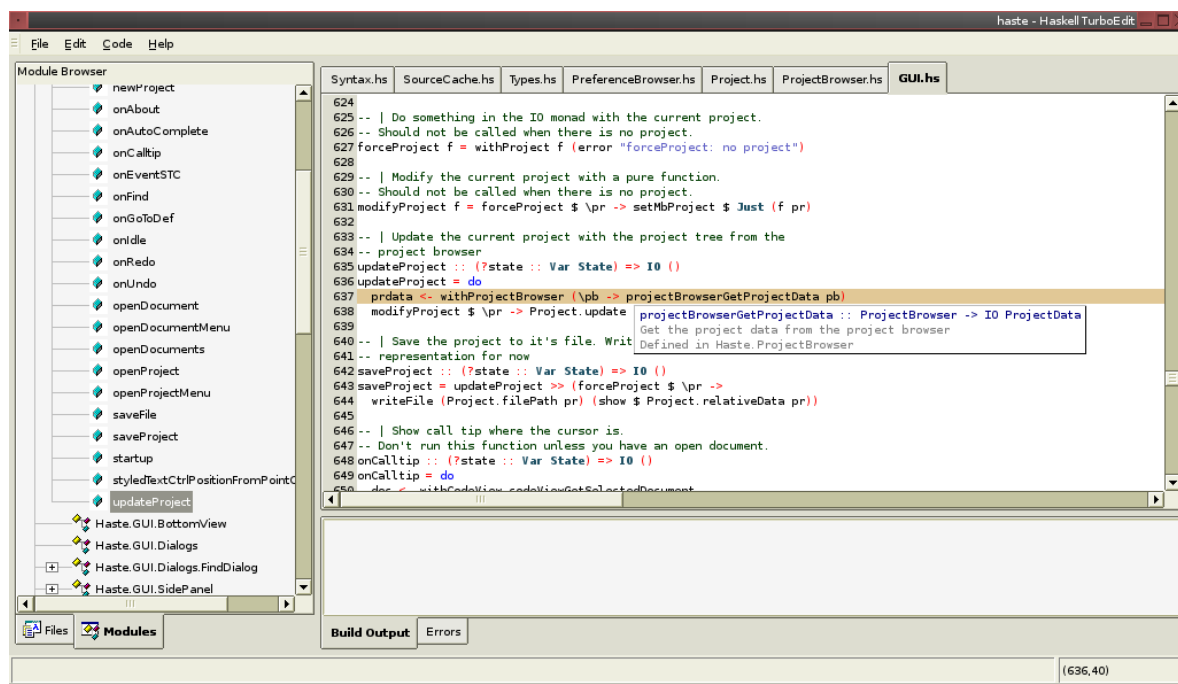


Figure 2.3: Haste (Haskell TurboEdit). It shows the module browser, and a tool-tip with type information and a Haddock comment

of work went into the communication between Haskell and a host as well, in this case Eclipse's plug-in architecture. This communication seems quite promising. However, we could not find any special features that do interact with GHC, apart from the syntax highlighter that uses a lexical analyser written in Haskell, and a console panel that loads GHCi, a GHC interpreter. The development of the plug-in is still in alpha phase, which puts things into perspective.

The Eclipse Haskell plug-in lacks features that, given that it can interact with GHC, are expected (and possibly will be present in future versions), such as the ability to show the type of an identifier in an interactive manner and name completion.

2.3 Haskell Editors Written in Haskell

It is a natural choice to write an editor for a specific language in that language because that is the language the developer knows well. The same holds for Haskell, and some attempts have been made to write a Haskell editor in Haskell.

Haste: Haskell TurboEdit

The Haste [BEK⁺05] IDE is the result of a student project from Chalmers University, developed in a few months by a small group of inexperienced Haskell programmers. It is written in Haskell and uses wxHaskell [Lei04] for the graphical user interface. Scintilla³ is used as the text editor in the IDE. Scintilla is a code-editing component with SciTE⁴ and Notepad++⁵ being its best known applications, and it already offers means for syntax highlighting, code completion and placing error indicators. It can show tool-tips in the editor as well, and hide segments of the source text. Scintilla has to be instructed where and how to apply these functionalities.

As can be seen in Figure 2.3, Haste contains a project browser with a virtual file system (file browser), a code view, and a compiler interface. The code view provides auto-completion for identifiers and types, and allows you to navigate to definitions. Furthermore it can show a tool-tip that

³<http://www.scintilla.org/>

⁴<http://www.scintilla.org/SciTE.html>

⁵<http://notepad-plus.sourceforge.net/>

contains the type information and Haddock [Mar02] style comments of an identifier. Next to the virtual file system is a module browser that lists modules with their functions and instances. The compiler interface lets you compile the code (possibly with separate file-specific compiler flags) with GHC. The output of GHC is parsed and shown in an error-pane, and error indicators are placed in the source code.

The authors want to display Haddock-style comments and type information in tool-tips, as seen in the figure. The initial plan was to use the GHC API for the parsing of the source and to retrieve type information. However, the GHC parser at that time discarded *all* comments, including Haddock-style comments. Therefore the authors abandoned the approach of using the GHC parser. Instead, a modified version of Haddock parses the code and retrieves type information.

As a students project, it wasn't finished or maintained after the course was over. One of the authors recently started with Haste2, but little is known about this project.

hIDE

Another IDE written in Haskell is hIDE [Sve02]. It doesn't come with a text editor of its own, but it has support for Emacs, Vim, and NEdit⁶. The philosophy behind this is that everyone has his own favourite editor, and why would you build something new when someone else has made something better. The features of hIDE include search, the creation of visual call and module graphs, and a tool to locate where a given function is called. Furthermore, hIDE has a module and file browser, it can generate a "to do" list based on comments containing TODO or FIXME, and it has CVS support.

The latest version of hIDE dates back to 2002, long before the GHC API was developed. Therefore it has no interaction with GHC and it is terribly out of date, which makes it almost impossible to build the project.

hIDE 2.x

More recently, some Haskell developers got the idea to create hIDE 2.x⁷. Despite the name, it has little to do with hIDE. The plan was to build an IDE, written in Haskell, with an extensible plug-in-based design and a tight integration with GHC. The editor itself would be based on Yi [SC05], which can be described as a Vim or Emacs-like editor written in Haskell and extensible in Haskell with plug-ins. Most features of hIDE 2.x had to become available through plug-ins, such as plug-ins for refactoring, revision control, and automatic indentation based on the syntax tree.

Unfortunately, the project was orphaned after nine months. After (undocumented) changes in the GHC API of version 6.8.1 the project also couldn't be compiled anymore.

Leksah

An IDE that *is* still being developed is Leksah [NF08]: an IDE for Haskell projects written in Haskell. The first beta release was earlier this year. It is based on GHC, GTK+⁸, and Gtk2Hs⁹, all of which are available on multiple platforms.

The editor contains a source view, a module or package browser, a type information pane and some other tools. All these panes can be moved around, put behind each other resulting in tabbed pages, split and merged. Figure 2.4 shows the IDE as it runs. Leksah uses the Cabal package system to manage the projects. Build errors are reported in a log pane, the locations are indicated in the source code, and the cursor is set to the location of the first error.

The source view can display *source candy*: user-definable combinations of ASCII characters can be represented by a different unicode symbol. For example `->` can become \rightarrow , and type variables can be displayed in Greek writing. This way,

```
x :: alpha -> beta
```

⁶<http://www.nedit.org/>

⁷<http://www.haskell.org/haskellwiki/HIDE/>

⁸<http://www.gtk.org/>

⁹<http://www.haskell.org/gtk2hs/>

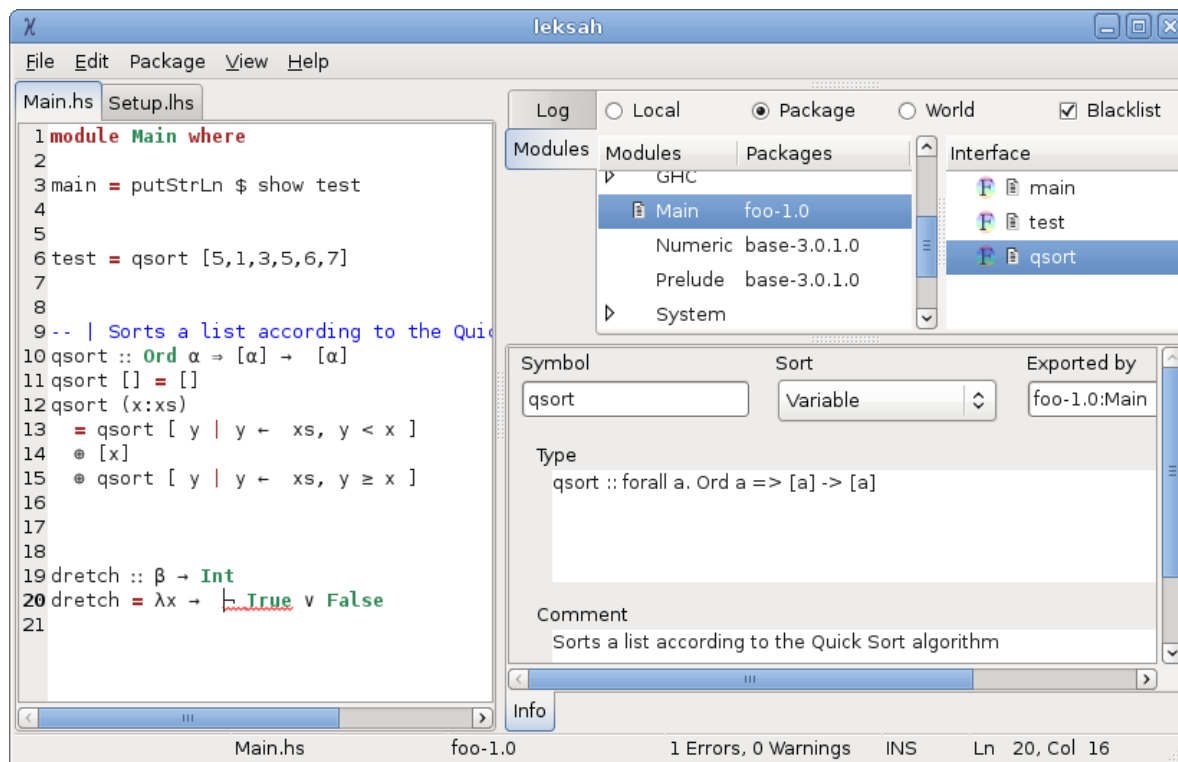


Figure 2.4: Leksah in action: source candy, package details gathered from `.hi` files, an error shown in the editor, and a Haddock comment at the type information

can be represented as

$$x :: \alpha \rightarrow \beta$$

The module or package browser shows all (local, global or in-package) modules and their exported items, which can be inspected in the type information pane. This type information and all package information is gathered by inspecting the Haskell interface (`*.hi`) files. Therefore only type information of exported items is available; type information of identifiers that are not in the export list of the module, or of local identifiers in `let` or `where` statements, is not available.

Leksah has the potential of becoming a practical tool to support the Haskell development progress. It has, however, no tight interaction with a compiler. Instead of using the GHC API, it relies on the static Haskell interface files to retrieve type information, and Leksah makes calls to external tools—Cabal—to build projects.

2.4 Other Editors or Tools

Vim and Emacs

Many other editors exist that aim at pure text manipulation, independent of the goal of the text (C, Haskell, \LaTeX , ASCII-art, ...). The Vim¹⁰ and Emacs¹¹ editors are favoured by many programmers and although they run just in a console and therefore may look simple, they offer a huge amount of features that makes it easier and faster to edit plain text, by reducing the number of keystrokes needed to perform edit operations. Vim and Emacs are much more mature and powerful than most editors (referring to the actual text typing component) found in IDE's.

For both editors—besides the usual syntax highlighting—a Haskell mode is available in the form

¹⁰<http://www.vim.org>

¹¹<http://www.gnu.org/software/emacs/>

of respectively a Vim script or a Lisp script. Both modes offer about the same functionality: the editor can navigate to imported modules, automatically finish the name of an identifier, and perform automatic indentation of the code. With only few keystrokes, the Haskell source can be built from within the editor. One can navigate to the position where a compilation error occurs, and after a successful compilation the type information for identifiers is available. The scripts start a GHCi session for compilation and analyse the output, making use of GHCi commands such as `:browse` and `:info`.

Of course this does not give a complete IDE and it is no real (or real-time) integration with GHC or any other Haskell compiler. Furthermore, the results are not always correct, like wrong syntax highlighting or indentation in special cases, when regular expressions are not as good as a lexical analyser. The features, however, that are provided by these editors and scripts are very powerful in aiding the programmer with his Haskell development. Both scripts are more useful than the Eclipse plug-in is at the current development stage. Vim and Emacs provide this functionality in only a few hundred lines of script code, on top of a very powerful editor.

HaRe

HaRe [LTR] is a Haskell Refactoring tool. It is not a real editor like all previous editors and IDE's, but it does manipulate and transform Haskell source code, using Vim or Emacs as a front-end. HaRe supports a set of refactorings: modifications to source code to improve its structure without affecting the functionality of the code. It is implemented in Haskell and makes use of the Strafinski [LV03] generic programming library and Programmatica¹², a Haskell development environment that supports stating and validating key properties. Programmatica has a parser and an API to access that parser, which is used to get the abstract syntax tree (AST) of the source code and to verify the correctness of the refactorings. Source code is refactored in HaRe by transforming the AST and printing the resulting tree. The printed refactored source code has to look like the original source code. However, the AST that Programmatica delivers contains no comments or source locations of the nodes. To retain the comments and source locations, HaRe works simultaneously with a token stream (to get comments and layout) and the AST.

The downsides of using Programmatica as the Haskell front-end (scanner, parser, scope analysis, etc.) is that it supports only Haskell98, and type checking can be slow. The former makes it less useful for practising programmers who use language extensions; the latter limits the HaRe developers to implement some refactorings that have to make use of the type system. Hence, when the GHC API became available, the authors of HaRe started a project [RT05] to port HaRe to the GHC API. The experience was that, although the Programmatica API and GHC API are very different, it is quite a straightforward task. The porting was not completed due to limited time. The authors like to see a higher level API for GHC that hides as much detail about the various state mechanisms used by GHC as possible.

Shim

The last tool we discuss here is Shim¹³. Shim is not an editor but a tool for editors. Shim starts up a GHC server to which other editors, such as Emacs and Vim, can connect. By using the GHC API it can provide features such as identifier completion, type lookup, and background compilation.

This 'client/server' separation seems very powerful in the sense that a lot of different editors and IDE's could benefit from the features Shim provides. Furthermore, because it is built on top of the GHC API, it has a very close connection with an external compiler. Unfortunately, at the time of writing there seems to be only an Emacs mode available that uses the Shim server, and the original developers website doesn't exist anymore, which indicates that the project may be stopped.

2.5 Conclusion

Many Haskell editor projects desire a tight integration with GHC, but almost all fail to provide it. Most editors have no connection with GHC at all; some do make external calls to GHC(i) and parse

¹²<http://www.cse.ogi.edu/PacSoft/projects/programmatica/>

¹³<http://code.haskell.org/shim/>

the output to use that somehow in the editor. This is not possible while one writes and there is no tight integration. Nonetheless, these editors can make the write-compile-debug cycle easier.

It seems that Visual Haskell is the only editor that is tightly integrated with GHC, as it queries the compiler and reports the results in the editor while one writes code. Unfortunately, it isn't actively developed or maintained anymore and it sticks with GHC version 6.6, while newer versions are already available. The large amount of overhead to communicate with the IDE framework (as with the Eclipse plug-in) makes the project difficult to maintain.

Chapter 3

Proxima

Proxima is a central subject in this thesis. This chapter gives an overview of what Proxima is, its architecture, and how to use it (Section 3.1). We also reveal a couple of details (Section 3.2) that form a basis for some topics later on in this thesis.

3.1 Overview

Proxima [Sch04] is a generic presentation-oriented structure editor. One gets a specialised editor suited for a certain type of documents by providing Proxima ‘sheets’. Different sets of sheets result in editors fit for different purposes. But what do the three adjectives ‘generic’, ‘presentation-oriented’ and ‘structure’ that precede ‘editor’ stand for?

Structure editor Proxima is aware of the underlying structure of the document. This allows the user to navigate over the structure of a document and perform edit operations on the structure, such as changing the order of sections in an article, or recursively downgrading a chapter to a section, and its sections to subsections.

Presentation-oriented editor In a presentation-oriented editor the user does not have to edit the internal document structure, as happens in many other structure editors. Instead, it allows the user to directly manipulate the presentation as shown on the screen. Whereas existing presentation-oriented editors are only applicable to a limited class of (mainly textual) documents, Proxima offers presentation-oriented editing on complex graphical presentations that may include derived values, such as a table of contents.

Generic editor A generic editor is suitable not just for a single document type, but for a large class of document types and can thereby replace all editors for these documents. Proxima, as a generic editor, delivers a uniform interface for these document types. A new editor has to be instantiated to support a new document type (as is explained further in this section), but building such a new editor requires only a fraction of the effort compared to conventional methods of building editors, due to the generic nature of Proxima.

Architecture

Proxima is under development at Utrecht University. Its architecture consists of six data levels with five layers in between, as can be seen in Figure 3.1. Each layer is responsible for a small part of the editing process, and each level is an intermediate result in the presentation process. This makes that the complexity of editing is divided in simpler subproblems.

The top-most level (document level) contains all the content and the complete structure of the document, but it has no exact details on how to render the data. At the other side is the bottom level (responsible for rendering) that gets instructed how to present (parts of) the content. This includes derived values (such as a table of contents) that are calculated in a higher layer. At the bottom level, however, not all information about the structure of the document is directly present.

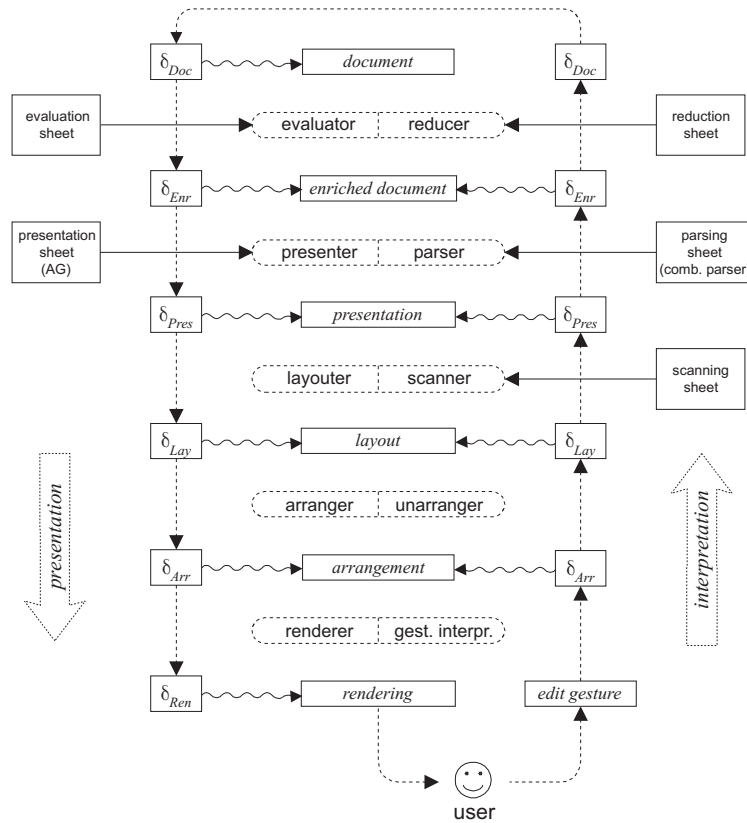


Figure 3.1: The layers of Proxima.

Between all adjacent levels is a layer that contains a bidirectional mapping, which maps one level to the other. This can be either to present a level (top-to-bottom direction) or to interpret a level after an edit operation is performed (bottom-to-top).

Basically two kinds of edit operations exist: *textual* edit operations and *structural* edit operations. After a textual edit gesture—typing some characters in the presentation—the rendering is scanned and parsed, which yields a new (enriched) document. On the other hand, a structural edit operation, such as inserting an element in a list by using mouse clicks, is performed *directly* on the (enriched) document, thus bypassing the parsing and scanning layers. A textual edit operation can be performed on a *parsable* presentation element, a structural edit operation on a *structural* element.

Extra State

Sometimes one wants to present only a part of the original document, for example to show only the table of contents in a word processor. The rendering level only contains the section headers, while the document still must have the contents of those sections. If a change is made in the table of contents at a lower level, this has to be interpreted. However, the content of the section, which is needed in the document level, cannot be inferred from this lower level. Therefore, the content must be stored internally. This information is called *extra state*.

More precisely, a layer maps one data level onto another, but the mapping in the reverse direction may not be able to compute all information for that level. That information has to be kept available in the form of extra state.

In the given example, data is omitted in the *presentation* direction while it is needed to interpret changes. The usage of extra state in the *interpretation* direction is whitespace information in source code, which is needed to render the document, but may have no meaning in the top-level document structure and therefore is not represented there.

Implementation and Instantiation

Proxima is implemented in Haskell and uses the GTK+ and Gtk2Hs libraries for the graphical user interface. To instantiate an editor, the basic document type of Proxima has to be extended and Proxima must be provided with a number of sheets that define the mappings between some of the data levels. For the presentation process (top-to-bottom) these mappings are an

- evaluation sheet, to compute derived values, and a
- presentation sheet, to define how to present the document

For the interpretation process (bottom-to-top), the mappings are defined in a

- scanning sheet, which contains regular expressions that describe the allowed set of tokens, a
- parsing sheet, to interpret the presentation level (inverse of presentation sheet), and a
- reduction sheet, to map edit operations on derived values onto edit operations on the document (inverse of evaluation sheet)

UUAG [SB04], an attribute grammar system developed at Utrecht University, is used for the presentation sheet. The UUAG compiler reads files containing a higher-order attribute grammar, in which semantic functions are described through Haskell expressions, and out of this description cata-morphisms ('folds') and data type definitions are generated. The presentations are written in the presentation language Xprez [SJ01].

The scanning sheet is (part of) an Alex¹ lexical syntax specification. Parsable tokens have to be defined; structural tokens are dealt with by the generic part of Proxima.

A parser combinator library from Utrecht University is used for the parsing sheet, with which parsable tokens can be parsed and structural tokens can be 'recognised'.

The rest of the sheets are Haskell files.

3.2 A Closer Look

Further in this thesis, we take a look at a couple of issues regarding the Proxima architecture or implementation. In this section, we look at some background information to understand those issues.

Whitespace

As mentioned under Extra State in the previous section, most whitespace (spaces, line breaks) in a source code editor has no meaning on the document level and is not represented. Nevertheless, it has to be presented when rendering the document. In Proxima this is implemented using a *whitespace map*: a (set of) mapping(s) from the *tokens* to the number of line breaks and spaces that follow the token, called *layout*. The focus (cursor) of the document is stored here as well. For example, the code fragment

```

1 2 3 4 5 6 7 8 9 10 11 12 13 14
x  y  =  ( 1  +  y  )
          *  3

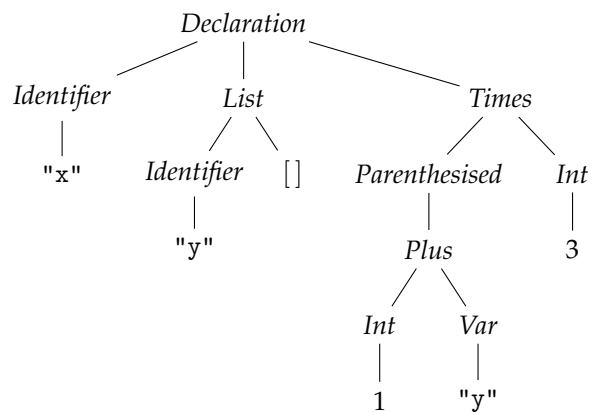
```

may be represented on the document level by

```

Declaration
(Identifier "x") -- function name
[Identifier "y"] -- patterns
(Times
  (Parenthesised
    (Plus (Int 1) (Var "y"))
  )
  (Int 3)
)

```



¹Alex: A lexical analyser generator for Haskell

In this example, 10 different tokens can be seen ('x', 'y', '=', '(', ...), each of which has its own layout: 'x' has no line breaks and 1 space, 'y' has 1 line break and 5 spaces, and '(' has no line breaks or spaces. The 'y' appears multiple times with a different layout. To uniquely identify the different tokens, all tokens have a presentation identity, or *IDP*. When the tokens get their IDP in order, the resulting whitespace looks like (Haskell pseudo code)

```

IDPx → (0,1)
IDPy → (0,2)
IDP= → (0,1)
IDP( → (0,0)
...
IDP) → (1,5)
...
```

The generic part of the Proxima *scanner* creates the IDP values automatically while scanning, identifies the layout for each token, and builds the whitespace map; the *parser* is responsible for storing the IDP values in the non-terminals it constructs. Once all tokens have an IDP, the whitespace map can be used to present the document. A structural edit operation, however, can introduce new nodes that have no IDP's stored automatically as the scanning and parsing layers are bypassed. In this case, the designer of the presentation sheet is responsible to assign fresh IDP's (depending on a given IDP counter) and update the whitespace map. Of course the next time the content of the editor is parsed, the newly inserted fragment is scanned and parsed as well, resulting in new generated IDP's.

Document Type

The data type that describes the underlying structure of Proxima must be defined in the *Document Type Definition*: a set of Haskell-like data type declarations that can contain primitive types (e.g. *Int*, *String*, *MyOwnDataType*) and lists (e.g. *[Int]*, *[MyOtherDataType]*), but no tuples, type variables, type synonyms, etc. Fields in the Document Type can be labeled with a name, and all constructors can declare a number of IDP's that are needed to present that constructor. The exact syntax can be found in Appendix A.

The Document Type for the example in the previous section, which consists of declarations with simple expressions, may be defined as follows:

```

data Document    = RootDoc    [Decl]
data Decl        = Declaration name : Id    patterns : [Id] Expr { idp : IDP }
data Expr        = Times      expr1 : Expr expr2 : Expr      { idp : IDP }
                  | Plus       expr1 : Expr expr2 : Expr      { idp : IDP }
                  | Int        Int                               { idp : IDP }
                  | Var        var : String                     { idp : IDP }
                  | Parenthesised Expr                          { idp1 : IDP idp2 : IDP }
                  | Let        decls : [Decl] Expr              { idp1 : IDP idp2 : IDP }
                  | Apply      expr1 : Expr expr2 : Expr
data Id          = Identifier  name : String                  { idp : IDP }
```

We can see that *Times* and *Plus* need a single IDP (to present the * and + respectively), a *Let* needs two (for the **let** and **in**), and function application (*Apply*) needs none, as it is represented by whitespace. In a future version of Proxima, the number of IDP's can be taken from the presentation sheet.

This Document Type Definition is read by a tool that generates the actual Haskell data types that Proxima uses. Furthermore it generates a UUAG file that is responsible for presentation-related administration, as well as function definitions that use the new data type. Among the generated function definitions are functions to read and write the document from and to XML, helper functions for the parse sheet, and functions that deal with structural edit operations. The generation tool extends the types from the Document Type with new constructors to represent *holes* and parse errors. A hole (or placeholder) is used, for instance, when a structural edit operation introduces a new node that has no value yet. The use of a list in the Document Type is 'desugared', which results in a new data type with a *Cons* and *Nil* constructor.

The *Expr* and *[Decl]* types are transformed as follows, omitting some details ($\langle \dots \rangle$):

```

data Expr
  = Times      IDP      Expr Expr
  | Plus       IDP      Expr Expr
  | Int        IDP      Int
  | Var        IDP      String
  | Parenthesised IDP IDP Expr
  | Let        IDP IDP List_Decl Expr
  | Apply                      Expr Expr
  | HoleExpr
  | ParseErrExpr (ParseError ⟨...⟩) deriving (⟨...⟩)
data List_Decl
  = List_Decl    ConsList_Decl
  | HoleList_Decl
  | ParseErrList_Decl (ParseError ⟨...⟩) deriving (⟨...⟩)
data ConsList_Decl = ConsDecl Decl ConsList_Decl
  | Nil_Decl                      deriving (⟨...⟩)

```

Of course the parse tree that the parsing sheet delivers must be of this generated type.

Chapter 4

Haskell Compilers

In this chapter, we discuss GHC in more detail (Section 4.1), as its API plays a significant role in the development of the work described in this thesis. We also take a peek at EHC, the Essential Haskell Compiler, under development at Utrecht University (Section 4.2). Both compilers are used in a Proxima editor instantiation.

4.1 GHC

GHC is the compiler of choice for most Haskell programmers. It produces fast code, comes with large set of extensions and is packed with an interpreter as well. As said in Section 2.1 on Visual Haskell, GHC offers an API that allows programmers to build tools, such as an IDE, on top of the GHC infrastructure, using the internal functionality of GHC. The GHC API was first introduced in 2005 with version 6.5 (a development version, with the odd suffix) and made its ‘public’ appearance in version 6.6, October 2006. GHC itself also uses the API to implement the different front-ends, as shown in Figure 4.1: GHCi is the interactive interpreter, `ghc --make` builds a multi-module Haskell program, and `ghc -M` generates dependency information. `one-shot`, not using the API, compiles a single module without chasing dependencies.

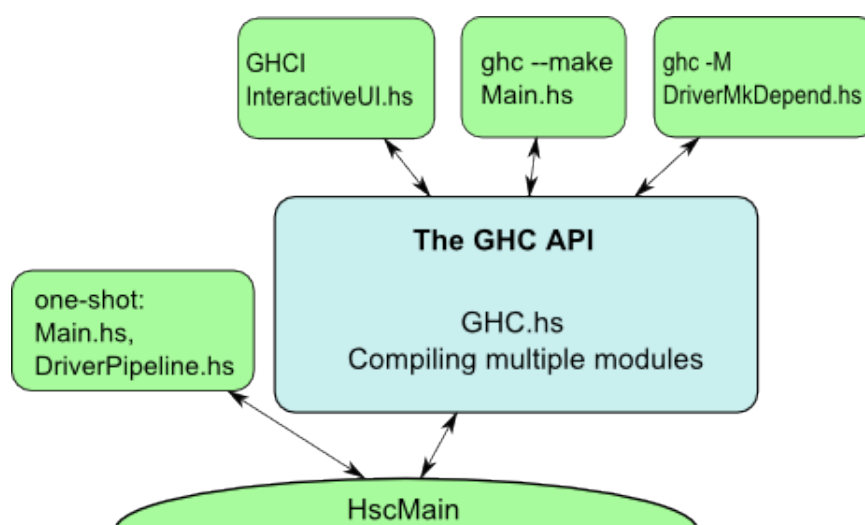


Figure 4.1: Global GHC front-end architecture

Architecture

The first stages of compilation (parsing, type checking) are the most interesting for this project. After a preprocessing phase (which, for example, interprets and removes `{-# OPTIONS ... #-}` pragmas, and ‘de-literates’ literate Haskell files), the `.hs` file goes through several passes. Only the first four passes can report errors or warnings to the user:

- Parsing, resulting in a *HsSyn RdrName* data structure
- Renaming, resulting in a *HsSyn Name* data structure
- Type checking, resulting in a *HsSyn Id* data structure
- Desugaring, resulting in a *CoreExpr* data structure

The *HsSyn* data type is very large, as it represents the entire syntax of Haskell in its full glory. The renaming and type checking passes add some information to the original parse tree. Therefore, the *HsSyn* data type is parametrised over the types of the term variables it contains. This results in more information over these terms in every pass without the need of introducing a completely new large data type. Approximately, these parameters have the following type: *RdrName* is a string, *Name* is a pair of a string and a unique identifier, and *Id* is a *Name* together with the type of a term.

The *HsSyn* data type includes, for example, parenthesis placements and the start and end character positions of expressions. Most comments are discarded by the parser, although some kinds of comments are present in the parse tree, such as Haddock comments.

The first three passes and resulting data structures provide the three hooks that can be used by an IDE to communicate with GHC. The desugaring process, which can only report pattern-match overlap warnings and errors when desugaring Template Haskell [SJ02] code quotations, is done *after* type checking, in contrast with most compilers, which first desugaring the parse tree and then perform the type inferencing on the simpler data type. The advantage of the late desugaring is that when an error occurs, GHC can display precisely the text that the user wrote.

The desugaring pass yields a data structure of type *CoreExpr*, which defines GHC’s intermediate language. It is very small: it consists of only nine constructors. After the desugaring pass the code generation begins, which is of no interest for this thesis.

Current Status

The API, introduced about two years ago, is far from finished. It is still in a state of flux with many changes of functions and function types through all GHC versions (currently v6.5, v6.6, v6.6.1, v6.8.1, v6.8.2, v6.8.3). The GHC API is very large and complicated. However, no Haddock [Mar02] documentation is available that explains how to use this complicated API. There are bits and pieces of comments in the source and there is a commentary¹ on the web that explains some of the global architecture and used data types. However, parts of the commentary are outdated when it comes to details. At this state, it is not easy to determine how exactly the various functions must be used, or how data structures must be initialised. Sources [RT05] from two years back say that there were plans for a higher level API that hides details and acts as an insulation against changes to the internal organisation of GHC. This, however, is yet not present and slows down the implementation of functionality that uses the GHC API.

4.2 EHC

EHC [Dij] is the Essential Haskell Compiler. It is not essential in the sense that it is the only one that matters, but instead it tries to capture the essence of Haskell and puts that in a compiler. Actually EHC is not a single Haskell compiler, but a whole series of compilers. Each compiler in the series adds some Haskell features or extensions to the previous compiler and is specified as a delta to its predecessor. The first variant only has explicitly typed lambda calculus, the second variant adds type inference, its successor extends it with polymorphism, and so on.

¹<http://cvs.haskell.org/trac/ghc/wiki/Commentary>

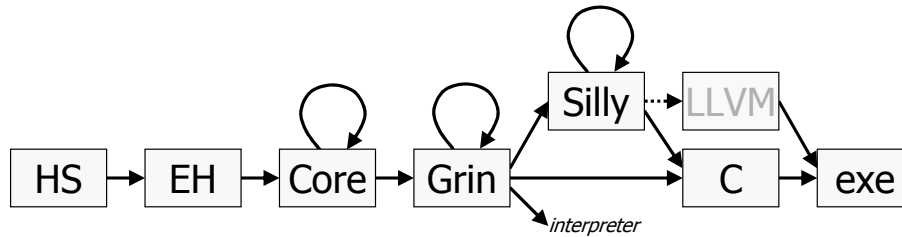


Figure 4.2: The EHC compilation pipeline

The compiler is not production-ready yet, but as it is developed internally at Utrecht University, all knowledge is close at hand and the barrier for feature requests is only one doorstep. EHC is being developed using a number of (Haskell-written) tools that produce Haskell code. Among these tools are the UUAG system (mentioned in Section 3). The parser is written with an error correcting parser combinator library. This means that the parser will always return a valid parse tree—even when parsing faulty code—that hopefully reflects the intention of the developer.

Architecture

Whereas GHC uses the same data type in the parsing, renaming and type-checking passes and then desugars the parse tree to a small core data type, EHC uses a more conventional compiler driver. The parser delivers a CST (concrete syntax tree) that contains the full Haskell syntax, on which name and dependency analyses can be performed. This is desugared to Essential Haskell (EH, Figure 4.2), which is a simplified version of Haskell and captures the essence of Haskell. This version is in its turn used for type analysis. The Essential Haskell data is then transformed to a core language, after which the code generation begins for a C or LLVM backend or an interpreter.

The CST from the parser contains information about most parentheses (but not for those in a class context, for instance), but comments are absent in the parse tree and token stream.

Because type inferencing happens *after* the desugaring process, it becomes difficult to map type information back onto the original parse tree. Furthermore, the source location information, which is a field in all nodes of the CST, is not always available, and therefore some parse or type errors are reported without a (correct) location. This makes it hard to find the relation between the CST and some errors reported by the type checker.

EHC does not provide an explicit API, but a standard installation of EHC does install a number of EHC packages that can be used to write your own compiler driver and access the internal data structures and functions of EHC.

Chapter 5

Motivation

Let us recollect the thesis question posed in the introduction (Chapter 1).

How can we link an external compiler to Proxima, such that the parser of the external compiler can be used as the Proxima parser, and such that Proxima can be used to create an IDE that has a tight integration with the compiler?

Because we focus on GHC and EHC, the mentioned IDE will be for Haskell. You may wonder,

- why yet another Haskell editor, and
- why use Proxima for that?

Why Another Haskell Editor?

We have seen in Chapter 2 that most of the Haskell editors that exist or that are under development are far from mature. Moreover they lack integration with a compiler that can enrich the coding experience. The single editor that is usable, Visual Haskell, is outdated, hard to maintain, and platform dependent. Concluding we can say that a good Haskell editor is needed.

Why Proxima?

A benefit of creating an editor as a Proxima instantiation is that Haskell programmers who need a Haskell editor, know Haskell better than other languages. Therefore, they can implement it easier and better in a Haskell environment, such as Proxima. Using Proxima also ensures that we do not have to deal with porting Haskell to another language, as seen with Visual Haskell and the Haskell plug-in for Eclipse.

Because GHC and Gtk2Hs, which are needed to build Proxima, are available on multiple platforms, including Windows, MacOS and Linux, Proxima and the resulting instantiated editor are multi-platform.

One can benefit from the unique features of Proxima as well, as it is a generic presentation-oriented structure editor. The information about the structure and semantics of the Haskell code opens opportunities for loads of useful features in a Proxima-based editor, including the possibility to rename an identifier only in a certain scope, source code folding that always works on the correct scope, displaying type information of all identifiers, the generation of type signatures, etc. Recent developments on Proxima made it possible for a Proxima instantiation to run in a browser, which opens the doors for a unique web-based Haskell editor.

Why GHC and EHC?

The decision to use GHC as a target compiler has practical roots: we want to create a product that can be used in the real world. With GHC we have a mature compiler with a lot of extensions and a large user base.

We look at another compiler as well, to be able to compare different compilers and to see how generic our approach is. For that we use EHC because it is developed in-house; thus, a lot of knowledge about it is close at hand and adjustments can easily be made—if necessary.

5.1 What To Expect in Part II

Part I discussed the preliminaries for this thesis and provided background information of Proxima, existing Haskell editors, and Haskell compilers.

Part II will discuss the problems that arise when using the parser of an external compiler as the Proxima parser, and what solutions are found. Of course the parsing process is looked at (Chapter 6), but the scanning process as well (Chapter 7). The minimal requirements of features that a compiler must offer, in order to be used as the parser of Proxima, are discussed in Chapter 8. The implementation of a framework to use the parser of an external compiler as the Proxima parser, as well as two Proxima instantiations that use GHC and EHC for parsing, is handled in Chapter 9.

Part II

Changing the World

*This Part shows a number of problems,
solutions,
and recommendations,
for both Proxima and compilers,
that deal with their communication*

Chapter 6

External Parsing

The next two chapters deal with the Proxima-specific issues that arise when using the parser of an external compiler as the Proxima parser. Parsing related issues are dealt with in Chapter 6, scanning related issues in Chapter 7.

The actual parsing comes with one problem: the parse tree of the external compiler must follow the Proxima Document Type format as explained in Section 6.1. To skip the parsing layer of Proxima has consequences that limit some features of Proxima, which is discussed in Section 6.2.

Bypassing the Scanner and Parser

The scanning and parsing layers form an integral part of the layered architecture of Proxima, and as such they are tightly interwoven with the system. To replace the Proxima parser by the parser of an external compiler is not just a matter of unplugging the Proxima scanner and parser and plugging the external one in: the parsing and scanning sheets (Section 3.2) form only a small part of the complete internal parsing and scanning system of Proxima, which is an essential part of the Proxima data flow. On top of that, Proxima is a presentation-oriented editor that can have *structural* elements, which are non-text like presentations of (parts of) the document. Such a structural element can be edited and has to be parsed as well, although it is not represented by text. This structural parsing is unsupported by conventional parsers.

Instead of replacing both layers, we try to incorporate the external parser in the sheets and let the architecture of Proxima remain untouched. We provide Proxima with a simple scanning sheet, discussed and justified in more detail in Chapter 7. The parsing sheet is simple as well: (lists of) structural tokens and (lists of) parsable tokens are parsed in turns. A semantic function concatenates the parsable tokens and pipes the result to the external parser; the structural tokens are by default ignored.

The approach of using two simple but general sheets needs no changes in the data flow and architecture of Proxima. We must be aware that, as seen in Section 3.2 under heading Whitespace, the scanner does not only tokenise the input, but also creates IDP's for the tokens, builds a whitespace map, and stores the location of the focus, and the parser stores the IDP's in the nodes.

The 'only' three difficulties are to

- call an external parser (Section 8.1) from a semantic function in the parse sheet,
- enable Proxima to use the parsed result of the external compiler (Section 6.1), and
- deal with the whitespace map, needed IDP's, and focus (Section 7.1)

6.1 Using Any Haskell Data Type in Proxima

In order to use the parser of an external compiler, Proxima must be able to access the data type that the parser returns, referred to as the *parser data type* hereafter.

The normal way is to provide a Document Type Definition (Section 3.2, Appendix A), from which a Haskell data type and helper functions that use that data type can be generated. This generated

data type is the *Proxima data type*. The Document Type Definition is very primitive, whereas a parser data type is more complex, at least that of GHC and EHC. The tool that generates the Proxima data type and helper functions, simply called the *generator* hereafter, can therefore not be used without modifications to either the generator itself—extending it to support full Haskell data types—or to the parser data type—‘downgrading’ it to the Document Type.

As said in Section 3.2, the generator adds data constructors for holes (to represent a placeholder) and parse errors to each data type, and it adds fields for IDP’s to each data constructor. It might be possible to *bypass* the generator if we add the hole and parse error data constructors directly to the parser data type by either using some generic programming techniques, adopting the data types à la carte approach [Swi08], or simply wrapping the data type in a wrapper type

```
data Wrapper a = Just a
                | Hole
                | ParseError
```

These approaches—if they even work, which is doubtful—come with a large implementation overhead with respect to the current Proxima system and solve only a small portion of the problem, since helper functions and an attribute grammar are generated as well.

It is more feasible to use the existing generator, possibly with adjustments. We can either extend the Document Type grammar and the parser of the generator to support full Haskell data types, or ‘downgrade’ the parser data type to be compatible with the current Document Type grammar.

Option One: Extending the Generator

Writing a parser for the generator that supports full Haskell data types is not that difficult, but the generator itself needs adjustments too to support that: Proxima data types must be generated correctly and all helper functions that work on the Proxima data type need modifications. This takes some effort to implement and will have a large impact on the generator and the way Proxima uses the generated types and functions. A slight problem is the UUAG system: currently it cannot handle type variables, thus this approach will not be fully compatible with Haskell data types and some instantiating with respect to type variables will be necessary.

Option Two: Downgrading the External Parser Data Type

The other possibility is to convert the parser data type to a Document Type Definition in a preprocessing step. With this approach, the generator and Proxima remain unmodified. The Haskell declaration

```
data Foo = F Int (Bool, [Foo])
```

is converted to the valid Document Type declarations

```
data Foo = F Int Tuple2_Bool_List_Foo
data Tuple2_Bool_List_Foo = Tuple2_Bool_List_Foo Bool [Foo]
```

Notice that `[Foo]` in `data Tuple2_Bool_List_Foo` doesn’t need to be expanded to `List_Foo`, as a Document Type Definition can contain lists.

Except for primitive types (*Bool*, *Int*, *String*, *Float*), there has to be a *mapping* from the *values* of the parser data type to values of the newly generated Proxima type as well. For the previous example, the mapping becomes

```
mapFoo (Parser.F int tup) = Proxima.F int (mapTuple2_Bool_List_Foo tup)
mapTuple2_Bool_List_Foo (bool,foos) = Proxima.Tuple2_Bool_List_Foo bool (map mapFoo foos)
```

Our choice

We chose the downgrading approach for two reasons. We want a solution that works without concessions, meaning that we want to either convert the whole parser data type and leave the generator unmodified, or keep the parser data type and make large modifications to the generator. The latter seems not possible because of the type variables. The second reason is that with the downgrading approach, we can leave the generator unmodified and do not have to worry about what impact a changed generator will have on the generic part of Proxima.

Conversion of a Haskell Data Type to the Proxima Document Type

Everything that is relevant to data types has to be taken care of when we convert a parser data type to a Proxima Document Type Definition.

Lists and tuples

Lists and tuples are handled as in the example: the occurrence of a list stays unchanged, but the name in other ‘complex’ types is changed (as in *Tuple2_Bool_List_Foo*). Tuples are *expanded*, or *desugared*, resulting in extra data types.

Parametrised types and type variables

A parametrised type has to be expanded as well, like the way the generator does with lists, similar to the example above. Thus, the occurrence of an *Either Int Foo* becomes *Either_Int_Foo*, where

```
data Either_Int_Foo = Left_Int_Foo Int
                  | Right_Int_Foo Foo
mapEither_Int_Foo (Parser.Left int) = Proxima.Left_Int_Foo int
mapEither_Int_Foo (Parser.Right foo) = Proxima.Right_Int_Foo (mapFoo foo)
```

Notice that lists and tuples are just sugared forms of parametrised types.

If a data type is parametrised with *type variables*, these have to be instantiated to all used variants. For example, if we have

```
data Bar a = B a
```

and *Bar* only occurs as *Bar Int* or *Bar Bool* in the rest of the data types, then we use *Bar_Int* and *Bar_Bool* in the data type and generate

```
data Bar_Int = B_Int Int
data Bar_Bool = B_Bool Bool
```

This is a workable solution if there are only few different instantiation alternatives for the type variables. Some data types, however, make use of wrapper-like data types. In GHC, for example, a *Located* wrapper is used, which contains a parsed node and the position in the source code that the node originates from:

```
data Located a = Located { theNode :: a           -- a node in a CST
                        , thePosition :: (Int,Int) -- the position of the node
                        }
```

The wrapper is used at almost all nodes and would generate a lot of expanded data types. It is more convenient to get rid of the wrapper constructors, as long as the extra information that the wrapper carries is not important in the editor. A *Located Expr* is replaced by just *Expr*, with a mapping function that removes the wrapper constructor

```
mapLocated_Expr (Parser.Located expr _pos) = mapExpr expr
```

Field labels, or records

A naive way to convert a Haskell data type with field labels (hereafter referred to as a *record*) is to ‘flatten’ it, such that the generator accepts it. Thus

```
data Zwoink = Z { a,b :: Int
                , c  :: Bool
                }
```

would become

```
data Zwoink = Z Int Int Bool
```

Unfortunately the field names get lost with this approach, whereas the Document Type *does* support them. In order to keep the named fields, it seems a logical step to convert the example above to the Proxima declaration

```
data Zwoink = Z a :: Int b :: Int c :: Bool
```

This has, however, the downside that this intermediate Proxima data type is not valid Haskell. For us it is desirable that it is a valid Haskell definition to make it compatible with other Haskell tools such as the *Language.Haskell* library, which is used in the implementation of the conversion tool.

We did not use the solutions above, but we chose to modify the parser of the generator, such that it can parse records. The parsed record can use the same data structure as a Document Type data type with named fields. It is only a small modification to the grammar of the Document Type and to the parser of the generator, while the data structure of the generator remains unchanged and thus no other internal changes are needed. The modified grammar is also shown in Appendix A.

Infix constructors and function types

Infix constructors

```
data X = Int :: Bool
```

or fields with a function type

```
data Y = Y (Int → Bool)
```

were absent in the parser data types we encountered thus far and not implemented in the conversion tool.

Furthermore, we cannot implement infix constructors using similar techniques as with lists and tuples because the conversion tool relies on the names of the data constructors, whereas the operator of an infix constructor cannot be part of a legal Haskell identifier.

A problem with function types is that a function field cannot be converted to any form of a primitive type without introspecting the function. The arguments of the function type may be dropped to end up with a primitive type, but then the mapping function has to instantiate the arguments, which is not always possible.

Type synonyms

Next, we have to deal with type synonyms. Every occurrence of a type synonym in a data type *or* in another type synonym is replaced by the actual data type, until we reach a fixpoint.

Newtype declarations

A **newtype** declaration is simply replaced by a normal data declaration.

Built-in type constructors

The built-in type constructors for lists, functions and tuples, such as

```
[ ]    Int           -- list constructor, sugared variant is [Int]
(→)   Int Bool      -- function constructor, sugared variant is Int → Bool
(,,)   Int Bool Bool -- tuple constructor, sugared variant is (Int, Bool, Bool)
```

have to be rewritten to the ‘sugared’ variants, after which any of the other conversions can be performed. The function constructor, as said before, is not converted. The trivial type `()` can be converted to *Unit*, where

```
data Unit = Unit
mapUnit _ = Unit
```

Qualified names

When we encounter a qualified type, such as *Data.Set*, we cannot simply drop the qualification (resulting in *Set*) because of the possibility of name clashes. To replace the dots in a qualified name with an underscore is also not favourable. Suppose the following new data type is generated:

```
data Tuple2_Foo_Bar_Meh = ⟨...⟩
```

The origin of this generated name is ambiguous. It could come from either one of

```
(Foo    , Bar.Meh)
(Foo.Bar, Meh    )
```

This name ambiguity problem does not exist for any of the previous methods that insert underscores, as long as the original (data) type names contain no underscores.

We decided to use double underscores for the qualified names, thus converting

```
Foo.Bar.Meh
```

to

```
Foo__Bar__Meh
```

There is still the possibility of name clashes, but it is less likely to happen than when we drop the qualification. The shown ambiguity problem is solved as well:

```
data Tuple2_Foo_Bar__Meh = ⟨...⟩
data Tuple2_Foo__Bar_Meh = ⟨...⟩
```

Class contexts, bang types, and derivings

Finally, class contexts (such as **data** *Ord a* \Rightarrow *Dretch a = D a*), bang types (such as *!Int*), and deriving declarations can be omitted during conversion as they are not needed anymore once they are in Proxima: a class context is obsolete since we instantiate all type variables, and the bang types and derived instances are used only internally in the external parser.

Implementation

A prototype of the conversion tool as described is implemented in Haskell and makes extensive use of the *Language.Haskell* library to parse, transform, and print the data types and to build the mapping functions. The tool gets as input the parser data type in a single Haskell module and a module that provides directions for the conversion steps, described below. A three-pass algorithm performs the conversion steps: first all type variables are instantiated, then most conversion steps are applied, and finally the type synonyms are substituted. The output of the tool consists of two files:

- The generated Proxima Document Type Definition
- A Haskell module that contains the mapping functions

The module that directs the conversion steps contains the instantiations for type variables, as well as some types and mapping functions of special cases that the editor designer needs, such as

```
-- FastString is used internally in GHC; Proxima likes to work with normal Strings
type FastString = String
mapFastString = unpackFastString
```


The prototype does not support infix constructors, function types and built-in type constructors, and the qualified of qualified names are simply dropped. Since these are all absent in the parser data types of GHC and EHC it forms no problem in this project.

6.2 Consequences of Using an External Parser

To use a general parsing sheet and to call an external parser has consequences regarding the features of Proxima, mainly because Proxima supports *structural* elements that need a structural parser, which can *reuse* parts of the previous document, and because the document structure can contain holes.

Structural Elements

One of the features of Proxima is that it can parse structural elements, such as a graphical representation of part of the document. Any conventional parser, however, accepts only string inputs. It is obvious that everything that has to be parsed externally has to be available in textual format, but this has consequences on the possibilities of using structural elements in the presentation.

As an example, we want to implement the possibility to hide some parts of the source code. The intended way to do that in Proxima is to present a structural element, such as a  graphic. The semantic function of the parser applies a *reuse* function, which reuses the node of the old document structure in the new document. However, we do not follow the Proxima paradigm of writing a full parser where the semantic functions can contain reuse functions; instead we only provide a basic parsing sheet that collects the parsable text from *parsable* elements and skips *structural* elements. As a consequence we cannot use all Proxima features as provided by the framework, but we can work around some features.

One solution is to let a structural element contain a set of textual tokens as extra state information, which forms the source code that the structural element represents. The Proxima parser extracts these tokens during parsing and mixes them with the parsable tokens. This approach, however, collides with the idea of extra state in Proxima: a structural element is a representation of part of the underlying document, whereas extra state is used for data that is *not* present on other levels. It is not hard to implement, but it changes the design of Proxima.

Another possibility is to perform an extra step before the source code is given to the external parser: for parsable elements we just take the string that it contains; when we encounter a structural element, we internally print the node of the document structure that it represents, which gives us a string that we can mix with the results of the parsable elements. This method is an extra layer of indirection before the external parsing can start.

Both solutions are not implemented and are regarded as future work.

Holes

A characteristic of structural editing in Proxima is that placeholders, or *holes*, can be placed in the document. A Proxima parser can recognise a hole and give one as a result. An external parser, however, cannot do that. By default, a hole is instantiated to `x`, after which the presentation contains `x` as well, instead of a hole. By instantiating a hole to `undefined`, the behaviour of a hole can be simulated in a Haskell editor, although the parsed output will show `undefined` instead of `{hole}`.

Incremental Parsing

External parsers do not support incremental parsing, whereas Proxima will in future versions.

Chapter 7

Scanning and Layout

One of the tasks of the scanning layer in Proxima is to facilitate the re-presentation of the original document by maintaining all whitespace in the whitespace map. Since we will bypass the scanning layer, we have to take care of this task ourselves. Section 7.1 describes how we take care of building this map and the related IDP's. Since most scanners of conventional parsers not only forget whitespace but comments as well, we have to take care of that too (Section 7.2); not presenting comments in the editor is unacceptable. We end this chapter with some remarks on preprocessing phases that are supported by some compilers (Section 7.3).

7.1 Bypassing the Proxima Scanner

In this section, we start out by listing the tasks of the Proxima scanner and giving some thoughts on the scanning process in this project. Then we will describe how the tasks of the Proxima scanner can be fulfilled when the parser of an external compiler is used. We distinguish *parser* tokens, from the scanner of the external parser, and *Proxima* tokens, as used in Proxima.

The function of an ordinary non-Proxima scanner is to tokenise the input to make it easier for the parser to parse the source code. The Proxima scanner, however, comprehends more tasks. The tasks of the Proxima scanner are:

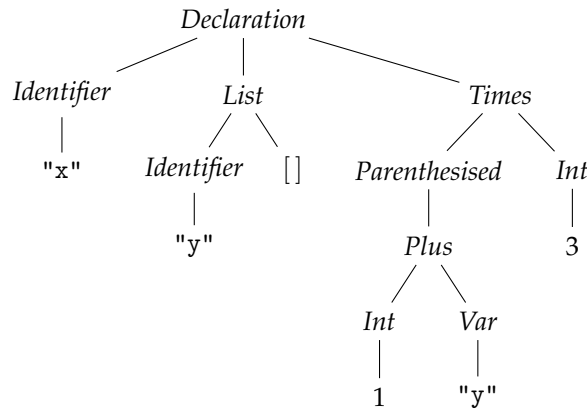
1. to provide input for the parser by tokenisation
2. to identify the layout of the Proxima tokens
3. to create IDP's and map them onto the layout (thereby building the whitespace map)
4. to store the current location of the focus in the whitespace map

The IDP's that the Proxima *scanner* creates are stored in the document structure by the Proxima *parser*. This is filed under item 3.

We do not want to care too much about the scanning process for the following reasons: since we use the parser of an external compiler that has a scanner of its own, we do not need the scanned tokens at all. Furthermore, if we write a full Haskell scanner, the scanner of the external parser gets duplicated and the editor becomes language dependent. It is no option to write a generic Proxima scanning sheet that tokenises the input text—a scanner that can be used in all editors that use the parser of an external compiler as the Proxima parser—because different programming languages and compilers have different sets of scanning rules.

The solution we propose is to use a very *simple scanning sheet* that contains a single scanning rule that, greedily, considers any sequence of characters as a single token. A token ends either at a structural element or at the end of the file. As a result of using this simple scanning sheet, no scanner is duplicated and the editors can be compatible with all languages and compilers. However, as a consequence we have to find other ways to perform the tasks of the Proxima scanner.

Next we will look at the four tasks of the Proxima scanner and how we can perform the tasks if we use the simple scanning sheet.

Figure 7.1: A parse tree of $x y = (1 + y) * 3$

Collect Input for the Parser

The first task of the Proxima scanner is to collect all input to provide that for the parser. The concatenation of the parsable tokens from the simple scanning sheet, which may be interspersed by structural tokens, must be identical to the text that the editor user wrote. The string that is the result of the concatenation can be passed to the parser of an external compiler because such parsers accept textual input and have an own scanner.

Identifying Layout

The layout of the Proxima tokens can be identified with the same approach as HaRe uses. HaRe makes use of the token stream of an external scanner to identify the whitespace between two successive parser tokens. For example, a parser token is represented by

```
data ParserToken = Token{ token :: String, startPos :: (Int,Int), endPos :: (Int,Int) }
```

which contains the line and column number of the character at the start position and of the character after the end position of the parser token. The code fragment

```
x y  = (1 + y)
    * 3
```

has the token stream

```
tokenStream = [Token "x" (1,1) (1,2), Token "y" (1,3) (1,4), Token "=" (1,6) (1,7)
              ,Token "(" (1,8) (1,9), Token "1" (1,9) (1,10), ..., Token "3" (2,8) (2,9)]
```

The layouts of x and y are identified by HaRe with

```
layout_x = layout tokenStream
layout_y = layout (tail tokenStream)
```

where

```
layout tokenStream = endPos (tokenStream !! 1) 'minus' startPos (tokenStream !! 0)
(a1,b1) 'minus' (a2,b2) = (a1 - a2,b1 - b2)
```

HaRe traverses over the parse tree *in-order* and takes the token stream into account in the traversal. Whenever a token has to be presented, its layout is identified using the token stream. Then the first element of the token stream is dropped to ensure that the next token to be presented is at the head of the token stream. Listing 7.2 illustrates the passing of the token stream for the given example, of which the data type is given in Section 3.2 and the parse tree is reprinted in Figure 7.1.


```

ATTR * [ | tokStr : [ParserToken] | ]
SEM Decl
  | Declaration
    name.tokStr    = @lhs.tokStr    -- The values of the tokStr attributes:
    patterns.tokStr = @name.tokStr  -- @lhs.tokStr = [tokenx, tokeny, token=, token(, ⟨...⟩, token3]
    expr.tokStr    =                -- @name.tokStr = [tokeny, token=, token(, ⟨...⟩, token3]
    tail $ @patterns.tokStr          -- @patterns.tokStr = [token=, token(, ⟨...⟩, token3]
    lhs.tokStr      = @expr.tokStr  -- @expr.tokStr = [ ]

```

Listing 7.2: AG-style example used in the parsing sheet: passing the token stream around

The token stream provides not only the source locations of parser tokens, but other source details as well, such as whether curly brackets and semicolons were used. This extra information is not present in a parse tree.

The token streams of the GHC and EHC scanners are accessible through their API and libraries, so we use the approach of HaRe to identify the layout. The layout is identified in the *parsing sheet* and *not* while presenting the document, because structural edit operations can make the token stream inconsistent with the document structure.

As we now can identify the layout of Proxima tokens, we need IDP's to store the layout in the whitespace map.

Creating and Storing IDP's, and Building the Whitespace Map

We need to *create* IDP's and *use* them in the mappings of the whitespace map. Furthermore, *storing* the created IDP's in the document structure is a task of the Proxima parser. Because we use an external parser, we have to take care of this as well. How to (1) create, (2) store, and (3) use IDP's is described in this subsection.

Creating IDP's All Proxima tokens require a unique IDP. Proxima provides a means to create fresh IDP's in the traversal over the document structure in the presentation sheet. Fresh IDP's may be necessary while *presenting* the document, because a structural edit operation may introduce a new presentation token, which requires a fresh IDP. We copy the idea of how fresh IDP's can be created and apply that in a traversal over the parse tree in the *parsing sheet*.

Storing IDP's The created IDP's have to be stored in the document structure, but where to store them? Usually, an editor designer defines IDP fields in the Document Type Definition. However, the DTD we use is generated (Section 6.1) and we do not want to add the IDP fields *manually* to generated constructors for two reasons: (1) it limits us from regenerating the file, and (2) in general you do not want to modify generated code. Neither can we predict the number of Proxima tokens that are needed for a constructor, so we cannot *generate* separate IDP fields specific for each constructor.

To overcome this, the conversion tool adds a *list of IDP's* field to each constructor, in which we can store all IDP's. It does not matter whether we have one IDP, two, none, or a variable number, because it all fits in a list.

As a consequence of using a list of IDP's, constructors do not have separate IDP fields anymore. This has a negative and a positive aspect: to pick the right IDP out of the list needs extra administration in the presentation sheet, which is negative. On the other hand, because the Document Type Definition does not contain explicitly separated IDP's anymore, which are only needed for presentation, the 'model' (DTD) and 'view' (IDP's) are separated, which is regarded good design.

As we now can create IDP's, have room to store them, and can identify the layout of Proxima tokens, it is time to put it all together by *using* the created IDP's and build a whitespace map.

Using IDP's: building the whitespace map Consider the example again, for which we have to create an IDP and store its layout in the whitespace map. The following code fragment can be added to Listing 7.2:

```
ATTR * [| whitespaceMap : WhitespaceMap |]
SEM Decl
  | Declaration
    loc.idps = [IDP_]
    lhs.whitespaceMap = Map.insert (@loc.idps !! 0) (layout @patterns.tokStr) @expr.whitespaceMap
```

For the = token, a unique IDP is created ($IDP_{=}$) and stored in a list of IDP's. This list will be stored in the IDP list field of *Decl* in the document structure. The corresponding layout is identified using the synthesised token stream attribute of *patterns*. The mapping from the created IDP to the layout is inserted in the synthesised whitespace map of *expr* (which, due to copy rules, subsequently used the whitespace map of its sibling). For other constructors we can create and store IDP's and build the whitespace map in a similar fashion.

Storing the Focus

The Proxima scanner is responsible for keeping track of the location of the focus or the selected region, which is stored in the whitespace map. The external parser does not provide this information, but this is not a major problem since Proxima has means to recover the focus when it gets lost.

Proxima can guess the location of the focus by using the pixel position of the focus before parsing. Whereas this behaviour is not always desired when editing a structural element because the element may reposition after parsing, it forms no major problem in a source code editor: we work mainly with textual data in a straightforward presentation in which elements do not move around as a result of parsing and re-presenting. Once the focus is recovered, it is stored in the whitespace map and poses no further problems for any following structural edit operations.

Summary

At the start of this section, we described the four tasks of the Proxima scanner. Summarised, the tasks and the solutions are

1. to provide input for the parser, by concatenating the tokens
2. to identify the layout, by using the token stream of the parser
3. to create IDP's, by using standard Proxima techniques, and to build the whitespace map, in a traversal over the parse tree
4. to store the focus, by using a recovery mechanism of Proxima

The solutions take care of the scanning process for all data that is present in the parse tree. However, data that is *not* present in the parse tree, such as comments and the source code before any preprocessing is performed, has to be taken care of separately.

7.2 Comments

Comments have no influence on the behaviour of the source code and therefore a lot of scanners (those of GHC, EHC and Programmatica, to name a few) consider most, but not necessarily all, comments in source code as whitespace, thereby making comments absent in the token stream. Nevertheless, comments have to be presented in a source code editor.

We propose three solutions to present comments in an editor. Problems arise with the first two solutions; the third is the implemented solution.

Modify the compiler For an editor of a specific compiler we can modify the scanner of the external compiler such that it puts comments in special parser tokens. However, this has impact on the parser of the compiler as well, which must be changed accordingly. We want the editor to be compatible with the compiler *out-of-the-box* without the need to ship it with a specialised version, which abandons this approach.

Comments scanner A simpler solution is to preprocess the source code with a ‘*comments scanner*’. This scanner can be written from scratch, or we can modify, for example, the GHC scanner such that it scans only comments. The comments scanner has to identify ‘*separate*’ comments in order to present them separately from the parse tree. With ‘*separate*’ we mean that the parse tree traversal in the presentation sheet does not encounter the separate comments and thus they have to be presented in another way.

Some problems arise with a comments scanner: for example, the scanner of GHC keeps Haddock comments and some pragmas in the token stream. These kind of comments are scanned by the comments scanner as well, and thus they are duplicated. Furthermore, a comments scanner limits the editors to languages with a Haskell-like comments syntax, although it only takes some small adjustments to this preprocessing scanner to support a different comments syntax.

The solution We come with a generic solution that does not suffer from the previous problems. We described how we use the parser token stream to build the whitespace map while traversing over the parse tree. If, at the same time, we walk over the original source text as well, we can see whether the source text between a parser token from the token stream and its successor contains characters other than whitespace. If so, a separate comment is identified that is not in the token stream. This approach is independent of the syntax and nesting rules of comments, and does not duplicate comments since comments that *are* in the token stream—‘*non-separate*’ comments— and which are probably in the parse tree as well, such as Haddock comments, are not identified as separate comments.

After we identified all separate comments, they have to be presented. For that, we can change the whitespace model of Proxima, such that it does not contain the number of line breaks and following spaces, but that it contains the actual strings that form the whitespace, *including* comments. With this modified model, the separate comments are an integral part of the whitespace map. Since all source code editors face the same problem with comments, this change to the design of Proxima is justifiable. It is, however, not implemented like this. Instead, we build a comments map that maps IDP’s onto lists of tuples of an IDP and the presentations of a comment:

```
type CommentsMap = Data.Map IDP [(IDP, Presentation)]
```

For the code fragment

```
x + {-comment 1 -}
    -- comment 2
    y
```

with IDP’s

```
IDPx IDP+ IDPcomment 1 IDPcomment 2 IDPy
```

the comments map looks like

```
IDP+ → [(IDPcomment 1, ⟨presentation of "{- comment 1 -}"⟩)
          , (IDPcomment 2, ⟨presentation of "-- comment 2"⟩)
        ]
```

When $+$ is being presented, IDP_+ has to be looked up in the comments map. The resulting presentations (with corresponding IDP’s) can be treated as normal presentation elements and are inserted in the current presentation.

7.3 Preprocessing

With GHC, the parsing takes place after a preprocessing phase (Section 4.1) in which literate Haskell files are turned into normal Haskell files, compiler option flags in pragma comments are interpreted, and macros are processed. After that, the (preprocessed) source is parsed. Other compilers have similar preprocessing phases. As a result of this preprocessing, the parse tree and the parser token stream leading to it possibly do not resemble the original source code anymore. After the deliteration of literate Haskell, the ‘normal’ text (text other than source code) cannot be located in the CST or parser token stream, and the parse tree doesn’t contain information about the C-macros that were present.

Because the preprocessing phase can *change* the layout, which makes that the parser tokens in the token stream can point to wrong locations in the original (non-preprocessed) source, we cannot handle preprocessing phases the same way as we did with comments. Therefore, preprocessing phases are not handled correctly in our editors and we do not support them. This also means that pragma comments, such as

```
{-# OPTIONS_GHC -fglasgow-exts #-}
```

are not interpreted by the parser and compiler, but such options can be passed separately to the compiler through the API.

A literate programming style can still be achieved by combining a Proxima source code editor with a Proxima word processor.

Chapter 8

Compiler API and Compiler Recommendations

In order to use the parser of an external compiler as the Proxima parser, we have to be able to access that parser. This chapter gives a set of minimal requirements that a compiler must comply to, in order to be able to be used as the Proxima parser (Section 8.1). Apart from parsing code, a compiler can provide useful information, such as type information. Section 8.2 describes some compiler features that can enrich an editor or that can make it easier to use the compiler for similar projects.

API's

We want to use the parser of an external compiler, as well as its type inferencer, to display type information and other information that a compiler can offer to enrich our editor. In open source compilers we have access to these features if we adapt the source code and make hooks available that provide the desired information. However, it has two downsides: finding the places where to get the right information may be difficult and the editor needs to be shipped with the specialised version of the compiler. This makes it hard to keep the editor maintainable (as seen on Visual Haskell, Section 2.1). Fortunately, some compilers provide an abstract programming interface (API), which insulates the internal workings of the compiler and hides it for the user (in this case a Proxima editor). This makes it easier to use the compiler for our purposes. Furthermore, the users of the editor do not need to use a specialised compiler. On top of that, an API tends to remain unchanged over different versions of a compiler, which makes the editor easy to maintain with respect to the compiler.

8.1 Minimal Requirements of a Compiler API

Both GHC and EHC offer hooks to access the intermediate compilation phases (discussed in Section 4), either through an explicit API (GHC) or through the libraries that come with the installation (EHC). To use the parser of an external compiler in our editors, the API has to support at least the following:

1. Access to the parse tree that is constructed from a string that contains the source code
2. Access to the token stream that is constructed from a string that contains the source code
3. Access to the data type definitions of the parse tree (needed for the conversion tool, Section 6.1)
4. Functions to inspect the tokens from the token stream (such as the start position of a token, or equality of tokens)

Some comments on the above list have to be made.

Ad. 1: parse failure If the parsing fails, we have to be informed about that (by returning *Nothing*, for example). Access to the parse errors is recommended as we want to be able to show where or why parsing failed. This is, however, not necessary for our editors to operate, although the use of the editor may be debatable.

Ad. 3: accessing the data type definitions To access the data type definition, the `:browse` command of GHCi can be used. The command prints the identifiers and data type definitions of a loaded module. Using this we have (albeit very indirectly) access to the data type definitions of GHC without needing access to the source code of GHC; a binary distribution of GHC is sufficient. The `browse` command of GHCi can also be used to get the data type definitions of EHC and closed source compilers.

Ad. 3: data type definition format As mentioned before (Section 6.1), the data type definition of the parse tree has some restrictions:

- All type variables must be able to be instantiated
- There must be no infix constructors or function types

8.2 Recommended API, or Desired Compiler Features

Apart from the minimal requirements of the API, there are also features a compiler can provide through its API that will enrich the editor. For example, we want to display parse and type errors in the editor, and for that the source location and type of the error (e.g. “wrong indentation”) must be accessible. If a type error is reported after the parse tree was desugared, it may be hard for the compiler to map the error back onto the right node in parse tree, but this mapping is outside the scope of this thesis.

Comments in the token stream Not only for this project, but for HaRe and other IDE’s as well, it is desirable to have direct access to the comments. It is difficult to place comments in a parse tree because it is undefined what piece of source code a comment belongs to and where in the tree it must be placed; Haddock comments form a special case and can be placed in a parse tree following special rules. Comments can, however, be present in the token stream as a comment token, just in order to preserve all data while scanning. A traversal over the parse tree together with the token stream can then be used to reproduce the original source code.

Word goes that GHC 6.10 will keep comments.

Type information A good IDE for Haskell code must be able to display type information. Therefore, the API has to provide functionality to access the results of the type inferencer. To let the type information be useful in an editor, it must be possible to map the information back onto the parse tree. Whereas GHC performs its type inferencing on the same data type as that of the parse tree, most other compilers, such as EHC, work on an abstract desugared version of the CST. There may be no direct mapping anymore from the type checked tree back to the parse tree, which makes it difficult to show the type information at the correct places in the editor. This problem, however, is compiler specific and not in the scope of this thesis.

For the purpose of this project—and probably for other IDE’s and programmers as well—it is ideal if the compiler supports *partially* type checked trees. This can be in the form of a *Maybe TypeInfo* field in the nodes that can contain type information, or by sending a query to the compiler with a request for the type information of a node.

Tokens in the parse tree The use of the token stream together with the parse tree to present the source is quite error prone. If accidentally one token too many or too few is taken from the stream, the whole remainder of the token stream is inconsistent with the parse tree, resulting in a malformed layout for the rest of the document. This can be due to a bug, but also if the presentation of a node is not implemented (yet), either because it is only used in a rarely used extension, or it is under

development. This is solved if the parser stores the tokens consumed to construct a node in a field of that node. Then, the token stream is not needed anymore, and the node itself has the information of its occurrence in the source code, such as whether optional curly brackets were used. It is only a relatively small and easy modification of the compiler.

It does, however, collide with having comments in the token stream, as there are no rules that define to what node a comment belongs.

Chapter 9

Implementation

A goal of this thesis project was to build two Haskell editors as Proxima instantiations (Section 9.7). The conversion tool as described in Section 6.1 was implemented too. The two editors are implemented using a domain specific language, which is introduced in Section 9.1 and Section 9.2 and described in detail in Section 9.3, Section 9.4 and Section 9.5. Section 9.6 describes how the display of comments is implemented.

We assume the reader is familiar with the UUAG syntax. Knowledge about the Proxima presentation sheet may help to understand some implementation details and examples.

Phi

For this thesis we implemented two Haskell editors as Proxima instantiations, which use the parser of either GHC or EHC as the Proxima parser. In the remainder, the generic part of the editors or the editors in general are referred to as *Phi*, standing for *Proxima Haskell Interaction/Integration/IDE*. The GHC and EHC editor instantiations are known as Phi_{GHC} and Phi_{EHC} , respectively.

In the previous chapters we described part of the modified *interpretation* process: in a traversal over the parse tree a whitespace map is built and IDP's are stored in the nodes of the tree. The presentation sheet contains a similar parse tree traversal. Both traversals are closely related if the source code is presented as text: first we read tokens from the token stream and create IDP's, then we *present* tokens *using* the IDP's. Therefore, we can use the presentation sheet in the presentation layer as well as in the parsing layer. A boolean flag determines the current process (interpretation or presentation), which, combined with lazy evaluation, ensures that nothing more is computed than necessary.

We will first look at an example from the presentation sheet of Phi_{GHC} to illustrate that a single sheet can be used in both directions without much extra implementation effort from the editor designer. The example shows the flow of the used attributes and gives a feeling for the implementation. Then we will describe the implementation of the generic part of *Phi* in detail. The generic part of *Phi* consists of the scanning and parsing sheets and the domain specific language used in the presentation sheet.

9.1 Example

Listing 9.1 shows a code fragment from Phi_{GHC} that is used for if-then-else expressions. The fragment consists of generated code and code written by the editor designer. The fragment is used in the interpretation direction as well as in the presentation direction. The most important part of the fragment for an editor designer is the function *makePres* and the list that it gets. The list contains *presentation commands*, written in a domain specific language, and *makePres* is an interpreter of that language.

We discuss flow of attributes in the example in the interpretation direction and in the presentation direction. The goal of *makePres* and the presentation commands in the interpretation direction is to build the whitespace map and store the IDP list in a field in the non-terminal. Whenever we refer to *the IDP list*, we mean that list. The IDP list has to be created, and during that process we distinguish two terms: the *parsed* IDP list, which refers to the IDP list that was stored in the interpretation process,

and the *created* IDP list, which refers to the (accumulated) IDP list that is being built at the moment. In the interpretation process, the created IDP list will eventually become the parsed IDP list. In the presentation process, the created IDP list is built as well, and it contains previously stored IDP's as well as newly created IDP that are a result of structural edit operations.

Interpretation Process

The goal of the tree traversal in the interpretation process is to build a whitespace map and store the IDP list in a field in the non-terminal. We will follow the attribute flow of the example in Listing 9.1 with this goal in mind. The running example we want to present is

```

<...> if True ≡ False
      then 1
      else 2
<...>

```

The call to *makePres* is

```
makePres [Token (keyword "if"), OTHER(condExpr), <...>] [] 6 wsMap tokStr True
```

where

```
tokStr = [token if, tokenTrue, token≡, tokenFalse, token then, token1, token else, token2, <...>]
```

The IDP list (@*idps*) is empty since the parse tree has just been initialised, and a goal is to fill it. The 6 represents an IDP counter to create new IDP's with. The *makePres* function handles the presentation commands in sequential order and starts with *Token (keyword "if")*.

The function *keyword "if"* identifies the layout at the head of the token stream and creates a new IDP, making use of the IDP counter. The new IDP and its layout are inserted in the whitespace map (*wsMap*), and an IDP list is created:

```
idpList = [IDPif]
```

Next, *makePres* passes the updated whitespace map, the created IDP list, and the tail of the token stream

```
tokStr' = [tokenTrue, token≡, tokenFalse, token then, token1, token else, token2, <...>]
```

to the next presentation command: *OTHER(condExpr)*.

The construct *OTHER* refers to a child node of *HsIf*, which subsequently applies *makePres*. The child node, *condExpr* in this case, must continue with *tokStr'*. This intermediate token stream is a return value of *makePres*. It is passed on to *condExpr* via the *condExpr.tokStr* attribute. The token stream is processed by *condExpr* and passed back to *lhs*:

```
tokStr'' = [token then, token1, token else, token2, <...>]
```

The command *OTHER(condExpr)* is responsible for passing the token stream of the child *condExpr* to the next command *Token (keyword "then")*; it uses *@condExpr.tokStr* to do that, as can be seen in the C-preprocessor macro definition *#define OTHER*. The previously passed on whitespace map and IDP list are forwarded to the command for "then" as well.

The presentation command for "then" is handled like *Token (keyword "if")*, but with two additional notes: (1) to create a unique IDP, an offset from the IDP counter is needed. This is simply the length of the created IDP list. (2) The new IDP is *appended* to the previous list of IDP's. The updated whitespace map is passed on, as well as the tail of the token stream,

```
tokStr''' = [token1, token else, token2, <...>]
```

and the updated IDP list

```
idpList' = [IDPif, IDPthen]
```

The presentation commands `OTHER(thenExpr)`, `Token (keyword "else")` and `OTHER(elseExpr)` are dealt with in similar ways. The passed on things of the last presentation command `OTHER(elseExpr)`, being the updated whitespace map, the tail of the token stream

$$tokStr''' = [\langle \dots \rangle]$$

and the created IDP list

$$idpList'' = [IDP_{if}, IDP_{then}, IDP_{else}]$$

form part of the output of *makePres*, and are stored in the local attribute `loc.phiPres`.

A generated semantic function processes `@loc.phiPres`: the whitespace map and token stream are passed to `lhs`; the created IDP list is stored in the node. The IDP counter is incremented (`loc.idpCounterOffset`) by the length of the created IDP list (as the parsed IDP list was initially empty).

Figure 9.2 schematically shows the attribute flow of the token stream and the whitespace map in the example. The whitespace map is copied downwards to *condExpr*, filled, and passed to its sibling. After *elseExpr* filled it, the whitespace map is used in *makePres* in *HsIf*, and afterwards passed to its parent. The token stream of the parent is used for the "if" token, passed to a child node, used for the "then" token, and so on. Finally it is passed back up to the parent.

Presentation Direction

The goal of the tree traversal in the presentation direction is to get a presentation. Because the token stream can be inconsistent with the document structure after structural edit operations have been performed, the token stream may *never* be used in the presentation direction. The distinction between whether we are in the interpretation process or the presentation process is indicated by the boolean flag `@lhs.isInterpretationProcess`.

We will again follow the call to *makePres*, but this time some attributes have changed.

$$makePres [Token (keyword "if"), OTHER \langle \dots \rangle] [IDP_{if}, IDP_{then}, IDP_{else}] \ 6 \ wsMap \perp False$$

The function *makePres* performs the presentation commands again in sequential order. The function *keyword "if"* gets the parsed IDP list as an argument (*idpList''*), in order to access *IDP_{if}*. It returns a presentation of "if" with the IDP.

The presentation is passed on to `OTHER(condExpr)`, after which it is prepended to the presentation attribute of the child of `OTHER`: `@condExpr.pres`.

$$pres = row [\langle presentation \ of \ "if" \rangle, @condExpr.pres]$$

For *keyword "then"*, an offset has to be increased in order to get the IDP from the correct position in the parsed IDP list. The returned presentation is appended to the previous presentation and forwarded.

The presentation commands `OTHER(thenExpr)`, `Token (keyword "else")` and `OTHER(elseExpr)` are dealt with in similar ways.

The resulting presentation, similar to

$$pres = row [\langle presentation \ of \ "if" \rangle, @condExpr.pres, \langle presentation \ of \ "then" \rangle, @thenExpr.pres, \langle presentation \ of \ "else" \rangle, @elseExpr.pres]$$

is stored in `loc.phiPres`, after which the ordinary Proxima attribute `loc.pres` takes over.

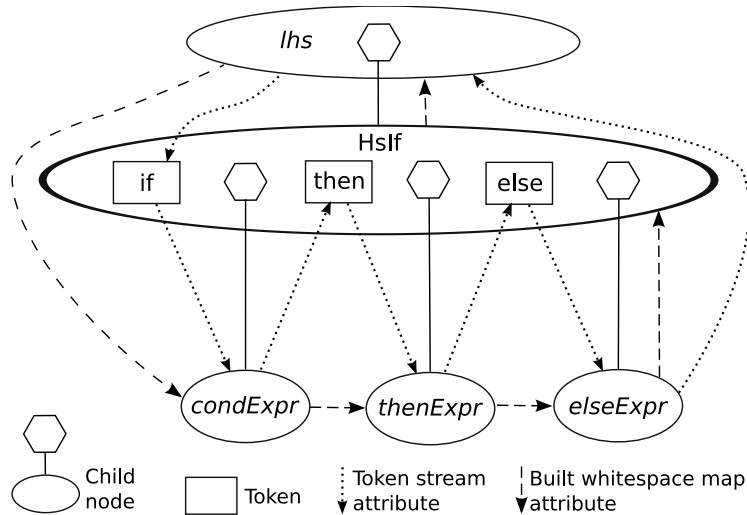
Justification

We mentioned that extra implementation effort from the editor designer is needed in the presentation sheet, due to the change from having separate IDP fields in the non-terminals of the Document Type Definition to having a single list of IDP's in the non-terminal that needs more administration. We can see in the example that the extra effort from the editor designer is small, especially if we consider that there is no need to write a parser, scanner, and Proxima Document Type Definition.

```

-- generated code:
#define ARGS(child) @idps @lhs.idpCounter @child.createdWhitespaceMap \
    @lhs.tokStr @lhs.isInterpretationProcess
#define OTHER(child) Other @child.pres @child.tokStr
ATTR * [ isInterpretationProcess : Bool
    | tokStr : TokenStream    createdWhitespaceMap : WhitespaceMap
    | phiPres : { (Presentation, [IDP], WhitespaceMap, TokenStream) }
]
SEM ** (loc.pres, loc.idps, lhs.createdWhitespaceMap, lhs.tokStr) = @loc.phiPres
    loc.idpCounterOffset = length @loc.idps - length @idps
-- editor designer written code:
SEM HsExpr | HsIf
    (loc.phiPres, [condExpr.tokStr, thenExpr.tokStr, elseExpr.tokStr])
    = makePres [Token (keyword "if")
        , OTHER(condExpr)
        , Token (keyword "then")
        , OTHER(thenExpr)
        , Token (keyword "else")
        , OTHER(elseExpr)
    ] ARGS(elseExpr)

```

Listing 9.1: Example code from the presentation sheet of Φ_{GHC} : if-then-elseFigure 9.2: Attribute flows of $tokStr$ and $createdWhitespaceMap$ in $HsIf$ in the interpretation process, corresponding to Listing 9.1. An ellipse represents the semantic function of a non-terminal. Square boxes represent *Token* commands. Hexagons, connected to other semantic functions, represent *OTHER* commands.

```

pCode = ( $\lambda$ tokens  $\rightarrow$ 
  let sourceCode :: String
    sourceCode = concatMap tokenValue tokens

    -- concrete syntax, token stream and errors from the external parser
    (maybe_cst, tokenStream, errors) = ExternalParser.parseString sourceCode

    -- convert the external parser data type to a Proxima Document Type
    enrichedDocument = RootEnr (case maybe_cst of
      Just cst  $\rightarrow$  cstToProxima cst
      Nothing  $\rightarrow$  parseErr $ ParsingParseErr <...>
    ) tokenStream errors

    -- create whitespace map and IDP's
    in create_idps_and_whitespaceMap enrichedDocument

) <$ pList pStructural <*> pList (pToken <*> pList pStructural)

```

Listing 9.3: Core of the parsing sheet as used in Phi

9.2 Generic Part of Phi

Four things form the generic part of Phi: (1) the data type conversion tool, the (2) scanning and (3) parsing sheets, and (4) a domain specific language with an interpretation function that is needed by the presentation sheet. The prototype implementation of the conversion tool is described in Section 6.1; the other implementations are described below.

Scanning and Parsing Sheets

The scanning sheet contains a single regular expression:

$$([\backslash x00-\backslash xff]) + \quad \{mkToken \$ \lambda s \rightarrow StringTk\}$$

This expression accepts all characters between two structural elements as a single token.

The parsing sheet parses a list of structural elements that is interwoven with parsable tokens. The structural elements are ignored; the values of the parsable tokens are concatenated to get the parsable text. The core of the parsing sheet is in Listing 9.3. The function *MyParserCaller.parseString* (indirectly) calls the external parser. The concrete syntax tree it returns has to be converted to the Proxima Document Type (*cstToProxima*). The function *create_idps_and_whitespaceMap* uses the presentation sheet to traverse over the parse tree in order to build the whitespace map and fill the IDP lists of the non-terminals in the document.

Presentation Sheet

An ordinary Proxima presentation sheet makes use of the Xprez combinator library, which provides basic building blocks to build complex presentations with. Some basic building blocks and combinator functions are

```

empty      :: Presentation
text       :: String  $\rightarrow$  Presentation
above, beside :: Presentation  $\rightarrow$  Presentation  $\rightarrow$  Presentation
col, row    :: [Presentation]  $\rightarrow$  Presentation
withColor   :: Presentation  $\rightarrow$  Color  $\rightarrow$  Presentation

```

The generic part of Phi defines a domain specific language that is needed to write the presentation sheet of a Phi instantiation. Phi provides an interpreter for that language, which is built on top of the

```

type Layout          = (Int, Int)
type WhitespaceMap    = Data.Map IDP Layout
type IDPCounter       = Int

type ResultType       = ((Presentation, [IDP], WhitespaceMap, TokenStream), ⟨...⟩)

makePres :: [PresCommand] → [IDP] → IDPCounter → WhitespaceMap → TokenStream → Bool
          → ResultType
makePres presCommands parsedIDPs idpCounter oldWsMap oldTokStr isInterp =

  let rec :: ResultType → PresCommand → ResultType
      rec x ⟨...⟩ = ⟨...⟩
  in foldl rec ((empty, [], oldWsMap, oldTokStr), ⟨...⟩) presCommands

```

Listing 9.4: The framework of the main combinator: *makePres*

Xprez library. The language and interpreter replace Xprez in most cases. The generic part of Phi also specifies an interface, which has to be instantiated.

Domain Specific Language

Listing 9.1 showed the use of the presentation commands *Other* and *Token*, which are part of our domain specific language. More presentation commands (**data** *PresCommand*) are available, and they can be split into three groups:

- single token commands (such as *Token*), which in the
 - *interpretation process* adjust the token stream, add IDP’s to the created IDP list, and update the whitespace map
 - *presentation process* present a token
- child presentation commands (such as *Other*), which drive the traversal over the tree
- miscellaneous commands, which includes the modification of other presentation commands (such as setting the colour of a token) and the administration of extra state information.

The *makePres* function helps to pass the token stream through the tree. In the interpretation direction the function makes sure that the IDP list is filled in the correct order, and in the presentation direction it takes care of finding the correct IDP that belongs to a token.

The arguments of *makePres* are: the list of presentation commands, the IDP list of the current non-terminal (empty in the interpretation direction), the IDP counter to create new IDP’s with, the whitespace map (which in the interpretation direction is being built and in the presentation direction is used), the token stream (only to be used in the interpretation direction), and a boolean flag that indicates in which direction *makePres* is used.

The function *makePres* accumulates a presentation, the created IDP list, and the updated whitespace map. This is performed by folding the presentation commands using the function *rec*. The token stream that is left over after all presentation commands are processed is returned as well by *makePres*. The basic framework of *makePres* with its types is printed in Listing 9.4.

We will discuss the three groups of presentation commands in detail. For each group, we show part of the *PresCommand* data type and explain the parameters of the constructors. We describe what the commands are used for and how to use them, and we give the implementation of function *rec* for each presentation command. We start with the most complicated group, the single token commands, and gradually fill in the ⟨...⟩ found in *rec* and *ResultType* in the sections describing the child presentation commands and miscellaneous commands.

9.3 Single Token Commands

The single tokens come in three different flavours: *Token*, *OptionalToken* and *Skip*.

```
data PresCommand = Token          PresFunction
                  | OptionalToken (String → PresFunction) ExternalParser.Token
                  | Skip
                  | ⟨...⟩
type PresFunction = Offset → [IDP] → IDPCounter → (Presentation, IDP)
type Offset = Int
```

What to Use Single Tokens for in the Presentation Sheet

Token The most used flavour is a *Token*, which contains a *presentation function*. In the interpretation process *and* when a new node was inserted by a structural edit operation, the presentation function creates an IDP given the IDP counter and an offset from the counter. In other cases, it looks up an IDP given the parsed IDP list and an offset in the IDP list. The distinction between the two cases is made by comparing the length of the parsed IDP list to the offset.

Two example *Token* presentation commands are

```
Token (operator "=")
Token (keyword "module")
```

where the functions

```
word, operator, keyword, symbol, literal, typeSignature, comment :: String → PresFunction
```

each define an own colouring scheme.

OptionalToken The *OptionalToken* is used for a token that is optional in the source code, such as optional curly brackets in Haskell, or parentheses that are not in the concrete syntax tree, as is the case with ‘deriving’ statements in the parse trees of GHC and EHC. The latter is illustrated by the following example, with the corresponding (simplified) parsed structures displayed in the comments.

```
-- two expressions, with varying number of parentheses
1 + 2                -- Plus 1 2
((1 + 2))            -- Parenthesised (Parenthesised (Plus 1 2))
-- two data declarations, with varying number of parentheses
data A = B deriving Show -- DataDecl "A" [Constructor "B" []] ["Show"]
data A = B deriving ((Show)) -- DataDecl "A" [Constructor "B" []] ["Show"]
```

The difference in the source code of using parentheses between the two expressions *is* reflected in the CST; the textual difference between the two ‘deriving’ statements, however, *is not* visible in the CST. Tokens, such as those last parentheses, of which the presence cannot be determined from the non-terminal or document structure, are considered optional tokens. The example also illustrates that an optional token may occur multiple times in succession.

The *ExternalParser.Token* parameter indicates the optional token and is used to compare tokens in the token stream. The parameter must be showable as its *show* is used as the argument of the *String → PresFunction* parameter.

An example of an optional token is

```
OptionalToken symbol (ExternalParser.CurlyBracketOpen)
```

Skip The last child presentation command is a *Skip* token, which presents nothing, but is used as a unit constructor. The use is illustrated in the following code fragment from `PhiGHC`, which presents an import declaration. The field `@isQualified` determines whether the import is qualified or not.

```
SEM ImportDeclaration | ImportDeclaration
    (loc.phiPres, ⟨...⟩) = makePres [ Token (keyword "import")
                                     , if @isQualified
                                       then Token (keyword "qualified")
                                       else Skip
                                     , ⟨...⟩
                                     ] ⟨...⟩
```

Implementation of Single Token Commands

Listing 9.5 shows a slightly simplified definition of *makePres* for the presentation commands *Skip*, *Token* and *OptionalToken*, which we will discuss.

Skip In case of a *Skip* command, a unit IDP constructor is added to the IDP list.

Token In the *Token* case, the function *presentToken* is applied to some arguments. It returns: a presentation of the token, the layout of the token (by making use of the token stream) with the corresponding IDP (by looking it up in the parsed IDP list, or creating a new one using the IDP counter if it cannot be looked up), and the tail of the token stream. The length of the created IDP list, *length idps*, is used as the offset from the IDP counter and as the position of the needed IDP in the parsed IDP list. Further on in this thesis can be seen that *length idps* does not always indicate the correct position in the parsed IDP list and a slight modification of the listed code will be described. Because of lazy evaluation, the layout and token stream are only identified and updated in the interpretation process. The presentation function parameter of *Token* is used to return the presentation and IDP of the token.

The new presentation and IDP are appended to the existing presentation and created IDP list, respectively. The new layout is inserted in the whitespace map, and the new token stream is passed on to the next presentation command.

OptionalToken An optional token can appear multiple times in succession. In the interpretation process, the number of occurrences of the optional token at the head of the token stream is counted by *countTokens*. Some scanners, such as the scanner of `GHC`, explicitly add optional tokens—curly brackets and semicolons—to the token stream, even if they are not present in the source text. In that case, the source location of such a token overlaps the following token, so it can be identified and *countTokens* removes it from the token stream.

To be able to *present* the correct number of optional tokens without using the token stream in the presentation process, the number of occurrences of an optional token has to be stored. We misuse an IDP for that, by storing the number of occurrences as an IDP. We store IDP ($-number_of_occurrences$) and restore (in $\lambda(IDP\ x) \rightarrow -x$) the negative value of the misused IDP to distinguish the misused IDP's from ordinary IDP's.

If the created IDP list is larger than the parsed IDP list, a structural edit operation must have added nodes to the document. In that case, we do not know how many times to present the optional token. It is a design choice to present the optional token once.

To actually get the presentation, the created IDP's, the updated whitespace map, and tail of the token stream that have to be the result of the optional token command, *foldl rec* is called recursively with *number_of_occurrences* times a *Token* command, which will accumulate the result.

9.4 Child Presentation Commands

The presentation attribute of a node contains not only single tokens, but the presentation attributes of child nodes as well. The *Other* production rule is used for that:


```

makePres _ parsedIdps idpCounter _ _ isInterp =
  let rec ((pres, createdIdps, wsMap, tokenStream), <...>) Skip
    = ((pres, createdIdps ++ [NoIDP], wsMap, tokenStream), <...>)

  rec ((pres, createdIdps, wsMap, tokStr), <...>) (Token f)
    = let (pres', (idp, lay), tokStr')
      = presentToken tokStr f (length createdIdps) parsedIdps idpCounter
      in ( ( row [pres, pres']
            , createdIdps ++ [idp]
            , Data.Map.insert idp lay wsMap
            , tokStr'
            )
          , <...>
          )

  rec ((pres, createdIdps, wsMap, tokenStream), <...>) (OptionalToken f token)
    = let (tokenStream'
          , number_of_occurrences
          ) = if isInterp
              then countTokens token tokenStream -- interpretation process
              else (⊥ -- presentation process
                    , if length createdIdps < length parsedIdps
                      then (λ(IDP x) → -x)
                        (parsedIdps !! length createdIdps) -- existing IDP: look it up
                      else 1 -- structural edit: new one
                    )
              in foldl rec
                ((pres, createdIdps ++ [IDP (-number_of_occurrences)], wsMap, tokenStream'), <...>)
                $ replicate number_of_occurrences (Token ∘ f ∘ show $ token)

  <...>
  in foldl rec <...>

calculate_layout :: TokenStream → (Layout, TokenStream)
countTokens :: ExternalParser.Token → TokenStream → (TokenStream, Int)

presentToken :: TokenStream → PresFunction → Offset → [IDP] → IDPCounter
              → (Presentation, (IDP, Layout), TokenStream)
presentToken tokStr presentationFunction offset idps idpCounter
  = let (layout, tokStr') = calculate_layout tokStr -- only in the interpretation process (lazyness)
    (pres, idp) = presentationFunction offset idpCounter idps
    in (pres, (idp, layout), tokStr')

```

Listing 9.5: *makePres* with *Skip*, *Token* and *OptionalToken*

```

data PresCommand = ⟨...⟩
                  | Other Presentation TokenStream
                  | ⟨...⟩

```

What to Use Child Presentations for in the Presentation Sheet

The token stream must be passed from the function *makePres* in the current node, to the child node, and back into *makePres* (Figure 9.2). To get it back into *makePres*, *Other* is parametrised by a *TokenStream* that is the resulting token stream of the child node. To pass the token stream to the child node, we set the token stream attribute of the child. The token streams *tokStr* are used as in

```

SEM ⟨...⟩ | ⟨...⟩
    (loc.phiPres, [child.tokStr], ⟨...⟩)
    = makePres [Token (symbol "1")
               , Other @child.pres @child.tokStr
               , Token (symbol "2")
               ] ⟨...⟩

```

In this example, *Token (symbol "2")* must use the synthesised attribute *@child.tokStr* to continue with the token stream of node *child*, which in turn uses the inherited attribute *child.tokStr* that is the result of processing *Token (symbol "1")*.

To be able to set the value of the inherited attribute *child.tokStr*, *makePres* returns a list of token streams¹ that contains some ‘intermediate’ token streams of the presentation commands. An intermediate token stream is the stream that is passed from one presentation command to another. An intermediate token stream is not always required, as we will show. Any two successive presentation commands seen so far match to one of these four cases:

1. first a single token command, then a child presentation command
2. first a child presentation command, then a single token command
3. two single token commands
4. two child presentation commands

In the first case, the child presentation command needs the intermediate token stream of the single token command. In the second case, the single token command gets the token stream via a parameter of the *Other* command. In the last two cases, the token stream is passed on directly, either from a single token command to a single token command, or using the copy rules of UUAG in the case of two child presentation commands. Thus, an intermediate token stream only needs to be returned when a single token command is directly followed by a child presentation command. To monitor this case, a boolean flag is used: single token commands set the flag to *True*, child presentation commands set it to *False*.

Implementation of Child Presentation Commands

The type of *ResultType* contains the list of token streams and the boolean flag, which fills in some part of *⟨...⟩*:

```

type ResultType = ((Presentation, [IDP], WhitespaceMap, TokenStream), [TokenStream], Bool, ⟨...⟩)

```

The *rec* functions for *Skip*, *Token* and *OptionalToken* have to return the two new arguments as well. The boolean flag is set to *True*, and the incoming list of token streams is passed on.

¹The current UUAG system cannot pattern match on a list, thus the actual implementation uses extra functions to flatten lists, like

```

makePres1 :: (a, [b]) → (a, b)
makePres2 :: (a, [b]) → (a, b, b)
⟨...⟩

```

The definition of *rec* in *makePres* for the *Other* command basically adds the presentation of the child node to the current presentation. If the previous presentation command was a single token command, indicated by the boolean flag, the previous token stream is appended to the token stream list so the child node can use that token stream. Otherwise the token stream list is simply passed on. The token stream of the child node, a parameter of *Other*, is forwarded to the next presentation command.

```

makePres _ _ _ _ _ =
  let <...>
    rec ( ( p1 , idps, wsMap, prevTokStr ), tss, wasToken, <...> ) (Other p2 tokenStream')
      = ( row [p1, p2], idps, wsMap, tokenStream' )
        , if wasToken then tss ++ [prevTokStr] else tss, False , <...>
      )
    <...>
  in foldl rec <...>

```

A Special Case of Child Presentation Commands: Token Separated Lists

Let us look at a simplified fragment from Phi_{EHC} , which is used to present data type declarations. For simplicity, the class context is omitted and the deriving statement is obligated, including parentheses.

```

SEM Declaration
| DataDecl
  (loc.phiPres, [ simpleType.tokStr, constructors.tokStr, derivings.tokStr ])
  = makePres [Token (keyword "data")
              , OTHER(simpleType)
              , Token (symbol "=")
              , <a command> constructors
              , Token (keyword "deriving")
              , Token (symbol "(")
              , <a command> derivings
              , Token (symbol ")")
              ] <...>

```

The presentation command *<a command>* is yet undefined. The child node *simpleType* represents the type of the data declaration, the child node *constructors* is a list of constructors, of which the elements have to be separated by vertical bars, and *derivings* refers to a list of derived classes, of which the elements have to be separated by commas.

The example

```

data Foo = X | Y Int | Z
  deriving (Show, Ord)

```

must get the parsed IDP list

```

parsedIdps = [token_data, token_ =, token_bar 1, token_bar 2, token_deriving, token_(, token_comma 1, token_) ]

```

The list separator tokens of *constructors* and *derivings*—*token_bar 1*, *token_bar 2*, and *token_comma 1*—are stored in the IDP list of *DataDecl*. In order to store the IDP list during the interpretation process, *constructors* and *derivings* have to return IDP lists, which contain the IDP's of the separator tokens.

To present *constructors* and *derivings* including the separator tokens, the children need to make use of the parsed IDP list of *DataDecl*. To access the correct IDP's in the list, an offset from the start of the list is needed as well; *constructors* must start at position 2 in the IDP list, *derivings* at position 6. A generated semantic function passes the parsed IDP list of *DataDecl* to *constructors* and *derivings*; the offsets are passed to the children by zipping the list of intermediate token streams and a list of offsets in the IDP list, which can be used as in:

```

SEM Declaration | DataDecl
  (loc.phiPres, [ (simpleType.tokStr , -)
                  , (constructors.tokStr, constructors.offset)
                  , (derivings.tokStr , derivings.offset)
                  ])
  = makePres ⟨...⟩

```

This has to result in

```

constructors.offset = 2
derivings.offset = 6

```

Next, consider a structural edit operation that appends a hole to the constructors, for which a new vertical bar is created as well:

```

data Foo = X | Y Int | Z | {hole}
deriving (Show, Ord)

```

Two related issues arise in this example.

First issue To present the first and second vertical bars, the IDP's are looked up in the parsed IDP list of *DataDecl* at the passed offset + 0 for the first bar and at the passed offset + 1 for the second one:

```

idpBar1 = (@constructors.offset + 0) !! parsedIdps
idpBar2 = (@constructors.offset + 1) !! parsedIdps

```

A similar approach for the third vertical bar is not valid, because

```

idpBar3 = (@constructors.offset + 2) !! parsedIdps

```

refers to *token_{derivings}*, whereas a new IDP must be created. To determine whether a new IDP has to be created, we must know how many IDP's for the separator tokens were created in the interpretation process, which in this case is 2.

Second issue To present the "deriving" token, we have to pick the IDP at position 4 in the parsed IDP list. In the implementation so far, this 4 was the length of the created, or accumulated, IDP list thus far. However, the created IDP list when presenting "deriving" is

```

idpListCreated = [tokendata, token=, tokenbar 1, tokenbar 2, tokenbar 3]

```

and *length idpListCreated* is 5, which refers to *token_{derivings}* instead of *token_{derivings}*.

Solving the issues Both issues are solved by storing the *number* of *parsed* IDP's used for separator tokens. As we then restore that number, the first issue is solved. The second issue is solved as follows: instead of relying on the length of the created IDP list, we accumulate an integer that refers to the current position in the parsed IDP list. In the example, the input value of the accumulated integer when presenting *Token* (*keyword* "deriving") was 2 (for *token_{data}* and *token₌*) and the restored number of parsed IDP's was also 2. By adding them up, we can lookup the correct IDP in the parsed IDP list

```

idpDeriving = (2 + 2) !! parsedIdps -- points to tokenderivings

```

Implementation of Token Separated Lists

A child presentation command that represents a token separated list uses the presentation command *List*, which gets two extra parameters that the *Other* command did not have: a list of IDP's of the separator tokens, to be stored in the accumulating IDP list, and the number of parsed IDP's.

```

data PresCommand = ⟨...⟩ | Other Presentation TokenStream
                        | List Presentation TokenStream [IDP] Int
                        | ⟨...⟩

```

The previous example fully filled in looks like

```

SEM Declaration | DataDecl
  (loc.phiPres, [(simpleType.tokStr, -), (constructors.tokStr, constructors.offset)
                , (derivings.tokStr, derivings.offset)
                ])
  = makePres
    [Token (keyword "data")
    , OTHER(simpleType)
    , Token (symbol "=")
    , List @constructors.pres @constructors.tokStr @constructors.idps @constructors.numIdps
    , Token (keyword "deriving")
    , Token (symbol "(")
    , List @derivings.pres @derivings.tokStr @derivings.idps @derivings.numIdps
    , Token (symbol ")")
    ] ARGS(derivings)

```

In the result type we can see the list of token streams zipped with the offset in the parsed IDP list, as is used by the child nodes. The accumulating integer that represents the current position in the parsed IDP list is present as well.

```

type ResultType =
  ((Presentation, [IDP], WhitespaceMap, TokenStream), [(TokenStream, Offset)], Bool, Int, ⟨...⟩)

```

In the interpretation process, we store the number of parsed IDP's using the same trick of misusing an IDP. This number is given by the child node as an argument of *List*: in the interpretation process it is the length of the returned IDP list; in the presentation process it is the restored value.

The accumulated integer, n , is incremented by the number of parsed IDP's + 1. The + 1 is because we add an extra IDP. All other *rec* functions increment n by the number of added IDP's, since the number is fixed in those cases.

We put n in a tuple with the previous token stream, as n represents the offset in the parsed IDP list. The *rec* function for *List* commands is similar to that of *Other*, with few changes, as highlighted:

```

makePres _ _ _ _ _ =
  let ⟨...⟩
    rec ( ( p1 , idps, wsMap, prevTokStr), tss, wasToken, n, ⟨...⟩ ) (List p2 tokStr' idps' offset)
    = ( (row [p1, p2], idps ++ (IDP (-offset) : idps'), wsMap, tokStr')
      , if wasToken then tss ++ [(prevTokStr, n)] else tss, False
      , n + 1 + offset, ⟨...⟩
      )
    ⟨...⟩
  in foldl rec ⟨...⟩

```

9.5 Miscellaneous Commands

The *makePres* functionality is completed with three more presentation commands: to increase the accumulated integer n , to modify a presentation, and to add an 'extra state' IDP.

```

data PresCommand = ⟨...⟩
                        | IncrOffsetBy Offset
                        | ModifyPres PresCommand (Presentation → Presentation)
                        | State Int

```

What to Use Miscellaneous Commands for in the Presentation Sheet

IncrOffsetBy The command *IncrOffsetBy* is used to increment the accumulated integer n . This is used in our presentation implementation for token separated lists: the presentation of the tail of a list uses the same semantic function and parsed IDP list as its parent, but it must use a different offset in the list.

ModifyPres Constructor *ModifyPres* is parametrised by another presentation command and a function that changes the presentation of the given presentation command. It is used for example to change the colour of the presentation of a presentation command. The modification function *mod* is part of the *ResultType*,

```
type ResultType = ( (Presentation, [IDP], WhitespaceMap, TokenStream)
                  , [(TokenStream, Offset)], Bool, Offset, Presentation → Presentation
                  )
```

ModifyPres is used in the following context:

```
SEM HsExpr | HsInfix
( loc.phiPres, ⟨...⟩ )
  = makePres [
                OTHER(leftHandSide)
                , ModifyPres (OTHER(operator)) ('withColor' operatorColor)
                , OTHER(rightHandSide)
              ] ⟨...⟩
```

In this example of an infix expression, only the presentation of the operator has to be presented in a different colour.

State To misuse the IDP list to store extra state information, the *State* command can be used.

Implementation of Miscellaneous Commands

All previous *rec* function definitions can *use* the modification function, and have to *return* a modification function. All occurrences similar to

```
row [previousPres,      addedPres]
```

have to be replaced by

```
row [previousPres, mod addedPres]
```

and thus applying the modification to the added presentation. In order to limit the scope of *ModifyPres* just to its presentation command parameter, the modification function that all *rec* functions from previous presentation commands return is *id*. *ModifyPres* is implemented with a recursive call to *rec*, applied to the new presentation command and the modification function inserted in the tuple.

For completeness, this is what the miscellaneous commands implementations look like

```
makePres _ _ _ _ _ =
  let ⟨...⟩
    rec (tuple, tss, wasToken, n, mod) (IncrOffsetBy m)
      = (tuple, tss, wasToken, n + m, mod)
    rec (tuple, tss, wasToken, n, _ ) (ModifyPres presCommand modify)
      = rec (tuple, tss, wasToken, n, modify) presCommand
    rec ((pres, idps, wsMap, ts), tss, _ , n, mod) (State m)
      = ((pres, idps ++ [IDP m], wsMap, ts), tss, True, n + 1, mod)
  in foldl rec ⟨...⟩
```

9.6 Comments

The functions *presentToken* and *calculateLayout* from Listing 9.5 are a bit more complicated than shown because they deal with comments as well. The token stream and source code are bundled because the original source code is used together with the token stream to build the whitespace map *and* to identify comments. To identify the comments, the function *calculateLayout* returns, apart from the layout and updated token stream, the actual source code between the first two tokens as well. This gives us

```
type TokenStream = (Parser.TokenStream, [String])
calculateLayout :: TokenStream → (Layout, [String], TokenStream)
```

The implementation of *presentToken* can be split into two cases: when in the interpretation process and when in the presentation process.

Interpretation Process of *presentToken*

In the interpretation process, the leading and trailing whitespace is trimmed from all strings that *calculateLayout* returns, in order to identify comments. For all lines of comments, an IDP is created and the layout is identified. A list of these IDP's and layouts, together with the IDP and layout of the token to present, is returned by *presentToken* in order to be inserted in the whitespace map.

The comments map, as defined in Section 7.2

```
type CommentsMap = Map IDP [(IDP, Presentation)]
```

is updated by inserting a mapping from the IDP of the original token to the IDP's and presentations of the comments. The type signature of *presentToken* changes accordingly:

```
presentToken :: TokenStream → PresFunction → Offset → [IDP] → IDPCounter → CommentsMap
              → (Presentation, [(IDP, Layout)], TokenStream, CommentsMap)
```

Presentation Process of *presentToken*

To present a token with the corresponding comments, the IDP of the token is looked up in the comments map. The comments that are looked up are appended to the presentation of the token (using the *row* combinator), and the resulting IDP's are returned to be used to increment the IDP counter.

Notes

Notice that if a token is not presented anymore (for example, when a node is replaced by a hole), its IDP gets lost and any comment that depends on that IDP disappears as well since it is not looked up in the comments map anymore. The same holds if comments were implemented using a modified whitespace model as discussed in Section 7.2. If we copy and paste a node (including the IDP's) the comments are copied as well.

Advantages of our implementation with respect to modifying the whitespace model are that Proxima can remain unmodified and that features such as automatic layout are a bit easier to implement when whitespace is represented by integers instead of strings. Furthermore, because the comments are separated from the whitespace map, updating the whitespace map when auto-layouting preserves the comments. However, an auto-layouted presentation may look different than expected when the comments are still present. This requires some further research.

A disadvantage is that we need to build a comments map and pass that around.

Movement of Comments

Consider the example where we have

```
x = [a, {-comment on b-} b]
```

and we insert a hole after *a*:

$$x = [a, \{-\text{comment on } b -\} \{hole\}, b]$$

The comment was bound to the IDP of the first comma in the list and therefore stays there, although it may have been desired to keep it close to the *b*. Nevertheless, it is in general undefined which node a comment belongs to (as it can appear at all whitespace locations) and whether it should have moved one comma further or not.

Experience With Adding Comments

When we implemented the comments in Phi, the major part of the combinator library and the two editors were already finished. Once we adjusted the *makePres* and *presentToken* functions in the generic part of Phi and adjusted the generator to generate some modified types and an extra attribute in the AG code, comments worked for both editors without any other adjustments. This shows us that the generic part of Phi is sufficiently abstract, and extra features can quite easily be added.

9.7 Phi_{GHC}, Phi_{EHC} and Proxima

The combinator library as described in the previous sections is used to implement Phi_{GHC} and Phi_{EHC}. Some issues we had with writing them are discussed in this section. Proxima needed some small adjustments as well, also described here.

Implementing Phi_{GHC}

In Section 4.1 we mentioned three different compilation stages of GHC (parsing, renaming and type checking) that use the same data type, but are instantiated with a different type for term variables. The parsing stage delivers *RdrName* term variables, which basically are strings representing identifier names, and the type checking stage delivers *Id* term variables, which basically are tuples of a name and its type.

This is a fragment of the GHC data type:

```
data HsSyn var = HsModule <...> [Decl var] <...>
data Decl var = <...>
                | SigDecl (Sig var)    -- type signature
                | BindDecl (Bind var)  -- binding
                | <...>
```

Whereas the GHC parser builds the complete *HsSyn RdrName* parse tree, the result of the type checker is *[Bind Id]*. The types are not only incompatible to form one parse tree (such as *Sig RdrName* versus *Bind Id*), the list of *bindings* from the type checker has to be inserted in the list of *declarations*.

To present type information, we initially used a library (which was written for Visual Haskell, and later updated for Shim) that looks up the type of an identifier given the line and column number. To get the line and column number, we needed the token stream in the presentation process, which is undesirable. Therefore, to be able to present type information in the editor, we do have to join the parse tree of GHC with the type checker output to get a full type checked parse tree.

By wrapping the term variables in a data type

```
data TermVar = Parsed RdrName | Typechecked Id
```

the parse tree can contain both *RdrName* and *Id* values and it stays type correct. If the source code passes the type checker, the mapping functions from the conversion tool replace the parsed bindings in the declaration list by the type checked bindings.

The question arises whether type information must be presented at all after a structural edit operation has been performed. For example, a 2-tuple can become a 3-tuple which makes that the type information in the tree is not correct anymore.

The presentation of a parse tree is usually built up in an in-order traversal over the tree. For example, to present

```
let x :: Int
    x = 1
    y :: Int
    y = 2
in x + y
```

we expect the non-terminal for the let-expression to contain a list of declarations that has the bindings and type signatures mixed together. However, all non-top level declarations, such as in a let statement, are represented in the GHC parse tree by two *separate* lists for the bindings and type signatures, which with a straightforward in-order traversal is presented as

```
let x = 1
    y = 2
    x :: Int
    y :: Int
in x + y
```

To get the correct interweaving, we introduce a data type

```
data Mixed = Binding GHC.Bind | TypeSignature GHC.Sig
```

and replace the two lists by a list of *Mixed* values. The mapping function (Section 6.1) for *Mixed* sorts the list by comparing the source locations of the binds and the type signatures.

Current Issues in Φ_{GHC}

There is a number of issues in the implementation of Φ_{GHC} , either because some parts are not implemented, or because of structural problems that may not be solved using the approaches described in this thesis and that are possibly due to bugs in GHC.

We didn't fully implement the presentation of kind signatures, generic classes, Template Haskell, the foreign function interface, Haddock comments, and some minor rarely used extensions. Using any of this in Φ_{GHC} will result in wrong layout and possibly a crash because the token stream is not updated. We expect no difficulties with implementing the presentations, but it will take some time.

GHC features that are not Haskell98 but *can* be presented contain parallel arrays, unboxed tuples, type and data families, and deprecation pragma's (but not in the module header).

Issues due to the GHC parser and scanner

Some issues in Φ_{GHC} are due to the parser and scanner of GHC.

Multiple arms, one binding The different arms for one binding are grouped together in a single node that contains the name of the binding, thus

```
f [] = <...>
f xs = <...>
```

is represented by the (simplified) declaration

```
Binding "f" [ <empty_arm>, <xs_arm> ] False
```

where

```
data Binding = Binding { name :: String, arms :: [Arm], isInfix :: Bool }
data Arm     = Arm     { patterns :: [Pattern], rightHandSide :: Rhs }
```

The node contains a flag to indicate whether the binding was written in infix or prefix notation. However, when infix and prefix notations are mixed, as in the legal Haskell fragment

```
[ ] ** [ ] = 1
( ** ) _ _ = 2
```

the flag is set to infix for the whole binding and thus for *both* arms—which obviously is not true—and `PhiGHC` presents both arms in infix notation. This also results in incorrect whitespace identification and an inconsistent token stream. We did not solve this problem and regard it as a GHC bug.

Infix binding, multiple parameters Because the prefix and infix written arms of a binding use the same data structure, the presentation of an infix written arm needs some tricky passing around of the token stream. For example, the fragment

```
(a *** b) c d = <...>
```

is represented by

```
Binding "***" [Arm [ <pattern_a>, <pattern_b>, <pattern_c>, <pattern_d> ] <...> ] True
```

The pattern list, however, cannot simply be presented the same way other lists are presented because of the operator that appears between the first two patterns, optionally followed by a closing parenthesis, and then the rest of the list. We solved this in our implementation, but not in an elegant way.

Deprecation pragma The deprecation pragma

```
{-# DEPRECATED x "foo" #-}
```

is one of the comments that is present in the parse tree and token stream. The scanner of GHC, however, considers `{-# DEPRECATED` as a single token. Therefore, we cannot use the token stream to calculate the whitespace between `{-#` and `DEPRECATED` and thus we use a hardcoded single space, although that may be different in the source code.

Hexadecimal values The GHC scanner converts hexadecimal values to the integer values, which means that `0xA` ends up as 10 in the token stream and parse tree.

Built-in functions The type checker of GHC removes the names of some standard functions from the parse tree, such as `length`. The example

```
x = sum [1..3]
y = length [1..3]
```

is, if it type checks, presented as

```
x = sum [1..3]
y = [1..3]
```

Due to limited time we have not figured out the exact reason why this happens.

Other Versions of GHC

`PhiGHC` is developed and tested with GHC version 6.8.2. As said in Section 4.1, the API is still in a state of flux, but we want to see how portable the editor is over different versions. According to the release notes for versions 6.8.2² and 6.8.3³ “there are some differences in the API exposed by the `ghc` package”, but only one or two minor changes must be made in the `Phi` module that uses the GHC API in order to let that module be compiled with GHC 6.8.1 or 6.8.3.

In the parser data type of GHC 6.8.3, two types are slightly changed: one type has an extra constructor in version 6.8.3, and a constructor of another type has one field less than in version 6.8.2. This is easily fixed, which makes that `PhiGHC` is almost compatible with newer versions of GHC. Version

²http://www.haskell.org/ghc/docs/6.8.2/html/users_guide/release-6-8-2.html

³http://www.haskell.org/ghc/docs/6.8.3/html/users_guide/release-6-8-3.html

6.8.1 has one type less than 6.8.2 and obtaining the data types was a little different, but other than that no changes had to be made. We can say that Phi_{GHC} needs modifications to be compatible with the different versions from the 6.8-branch of GHC, but the changes are small.

The version that precedes 6.8.1 is 6.6.1, in which the API was fairly new and underdeveloped. Some changes are needed in the module that uses the GHC API, but those are easy to fix. However, there are a lot of small differences between the parser data types of both versions, which makes it a considerable amount of work to let the presentation sheet of Phi_{GHC} be compatible with the data type of GHC 6.6.1. Therefore we conclude that Phi_{GHC} is not compatible with GHC versions lower than 6.8.1.

Implementing Phi_{EHC}

As opposed to Phi_{GHC} , Phi_{EHC} can display the complete parse tree of the external compiler.

By default, EHC uses an error correcting parser that always returns a valid parse tree, even if the code contains parse errors. This behaviour is undesirable in Phi because we do not want an editor to change the source code by itself. Once we knew what to look for, it was easy to write our own small compiler driver that did not have the error correcting behaviour.

The basic implementation of Phi_{EHC} was a straightforward task without the difficulties we encountered in Phi_{GHC} ; the hardest part was to figure out what the different constructors stand for and how they have to be presented.

Phi_{EHC} does not offer any extra features (apart from displaying the parse errors in the editor) because the implementation was only meant to show that the generic part of Phi is abstract enough to support different compilers. Furthermore, the developers of EHC state that because the desugaring process takes place before type checking, it may be hard to map the results of the type inferencer back onto the parse tree, in order to display types in the editor. This is, as said, outside the scope of this thesis.

Changes to Proxima

Only few modifications are made to Proxima. The type of the parsing sheet was changed such that the Proxima parser can return the built whitespace and comments map as well, in order to get them into the Proxima framework. The presentation sheet has an extra parameter in the form of the comments map. A couple of functions that use the parsing and presentation sheets had to be modified as well to support the changed types, but that was a trivial task. Other than that, Proxima remains unchanged.

Part III

Conclusion & Appendices

"A conclusion is the place where you got tired of thinking"
— Arthur Bloch

Chapter 10

Conclusion

In the introduction of and motivation for this thesis (Chapter 1 and Chapter 5) we posed the question

How can we link an external compiler to Proxima, such that the parser of the external compiler can be used as the Proxima parser, and such that Proxima can be used to create an IDE that has a tight integration with the compiler?

We focussed on using GHC and EHC as external compilers, which resulted in two editors for Haskell called Phi_{GHC} and Phi_{EHC} . We tried to keep Proxima unmodified as much as possible to let Phi be compatible with the future improvements of Proxima, and therefore most was implemented in the Proxima sheets.

To bypass the parser and scanner of Proxima yields three main problems, as discussed in Chapter 6: the external parser must be called from a semantic function in the parse sheet, Proxima must be able to use the parsed result, and since we do not use the special features of the Proxima scanner, the whitespace map must be built together with the corresponding IDP's.

To make a call to the parser of an external compiler, the compiler has to offer a means to access its internal functionalities, preferably without making changes to the source code of the compiler. Section 8.1 gives us a number of minimal requirements that the API of a compiler must offer, in order to be able to be used as the Proxima parser. The API of GHC and the libraries that come with EHC comply to the minimal requirements and they can be used, as shown in Section 9.7, as the parsers of Proxima.

Section 6.1 showed how most data types and data structures of a parser can be converted to a Proxima Document Type Definition. Although it is not a complete solution because function types and infix constructors are not supported and name clashes can occur with the generated names, it works for all parser data types that we have seen.

Finally, we demonstrated how the whitespace map can be built by using the token stream of the scanner of a compiler (Section 7.1), and how to create and manage (Chapter 9) the corresponding IDP's. To manage the IDP's, some careful administration is needed, but that is mostly abstracted over by a domain specific language and an interpreter for that language. The extra administration can be justified as well because we do not need to manually add IDP fields to the Document Type Definition, and it separates the IDP's ('view') from the Document Type Definition ('model'). Notice that the whitespace map and corresponding IDP's are only needed to re-present the input source text; any other view of the parse tree can be implemented in the ordinary Proxima way, without making use of the domain specific language.

For text that is not represented in the token stream, such as a comments, we provided a generic solution to present it in the editor, shown in Section 7.2.

We can answer the first part of the thesis question,

[can] the parser of the external compiler [...] be used as the Proxima parser[?]

with "yes" for the fact that the result of the external parser can be used as the Proxima document structure, but there are two notes. First of all, any preprocessing (Section 7.3) that the compiler performs is not supported because the original input text and the parser output can be inconsistent. Furthermore, some of the features of the Proxima parsing system cannot be used (Section 6.2), such as the parsing of holes and reusing part of the previous document.

The second part of the thesis question,

[can] Proxima [...] be used to create an IDE that has a tight integration with the compiler?

cannot be answered with a convincing “yes”. In Chapter 9 we discussed the implementation of our editors, but we have also seen with GHC that the parse tree of a compiler is not formatted ideally to be used for a *textual* re-presentation. Think about bindings and type signatures in a `let` statement that are represented by separate lists, or the issue that a boolean flag determines whether infix notation was used instead of using different constructors—as how it is represented in EHC. The example in Section 9.3, that showed us that the parentheses around *Show* in the **deriving** declaration are not represented in the parse tree, supports the claim, and the list of known bugs due to the parse tree being designed for compilation instead of presentation is only growing.

We can say that an ordinary parse tree—used for compilation—is not perfectly suited to be a Proxima parse tree—used for presentation.

To be able to display the type information of a selected identifier we need fields that contain the type information. The GHC parse tree contains these fields, but the document structure of Φ_{EHC} does not. It is not an option to map the type information onto the parse tree *on the fly* while presenting, because structural edit operations can make the Proxima document structure inconsistent with the state of the type inferencer of EHC. Therefore, the fields must be added to the document structure. However, adding extra fields to the document structure is not feasible, because the Proxima Document Type is generated.

We conclude that the data type that an external parser returns is not perfectly suited to form a Proxima data type. Whereas most problems can be solved with creative programming, some problems still exist that make the editors unusable for any production programming. The support for structural editing in Proxima adds a layer of complexity that has to be dealt with, which makes that we cannot rely on some basic assumptions that an API relies on, such as that line and column numbers are always available. This combined, we must say that the goal to create an IDE as a Proxima instantiation that has a tight integration with an external compiler, failed.

10.1 Future Work

In this thesis we mentioned that some parts are regarded future work. It is summarised here. We also compare Φ with HaRe and make some remarks on Proxima.

Future Work Mentioned in this Thesis

Section 6.2 mentioned that we cannot reuse parts of the previous document when parsing because the external parser needs the full input text. One of the possible solutions was to present structural elements internally in order to retrieve the source text that the structure represents, and feed that text to the parser. This extra step is not implemented and some further research is needed to see how feasible it is. Nevertheless, this functionality is necessary to be able to pass any structural representation of the source code to the parser.

We also mentioned future work in Section 9.6 regarding what to do with comments after auto-layouting, or after structural edits in general.

Combine Φ with HaRe

Some of the features we want our editors to support are related to refactoring, such as replacing an identifier by another one in a certain scope. This resembles the Haskell Refactoring (HaRe) project, as HaRe and our editors both apply transformations over the parse tree and display the resulting trees similar to how the user wrote it. To avoid implementing refactorings that are already implemented, it can be interesting future research to combine HaRe and Φ and end up with a presentation-oriented structure editor for Haskell with good refactoring support.

Proxima Specific Future Work

Because a Proxima Document Type Definition can contain IDP fields, which are only used for presentation, the model and view of Proxima are not separated. This thesis provided a means to separate both, although it is a quite specific solution for Phi; nevertheless, the separation has to be in the general framework of Proxima as well.

On the other hand, our experience is that the parsing and presentation sheets are so closely related that both have to be written almost simultaneously in order to get the one-to-one mapping between structural and parsable elements correct, at least when the editor designer is not very familiar with Proxima. A Proxima editor in which both sheets can be written simultaneously may be a useful tool to design the parsing and presentation sheets.

A useful extension of the Document Type Definition is to add a *Maybe* data type. It is commonly used in parser data types, and the UUAG system supports *Maybe* declarations natively as well.

Appendix A

Document Type Grammar

The grammar for the Proxima Document Type, with $\langle document \rangle$ as root, is as follows

```
 $\langle document \rangle ::= \langle dataType \rangle^*$   
 $\langle dataType \rangle ::= \text{data } \langle upperId \rangle = \langle prod \rangle (| \langle prod \rangle)^*$   
 $\langle prod \rangle ::= \langle upperId \rangle \langle field \rangle^* idps?$   
 $\langle field \rangle ::= (\langle label \rangle :)? \langle type \rangle$   
 $\langle label \rangle ::= \langle lowerId \rangle$   
 $\langle type \rangle ::= \langle upperId \rangle | [ \langle upperId \rangle ]$   
 $\langle idps \rangle ::= \{ \langle field \rangle^* \}$   
 $\langle upperId \rangle ::= \langle upper \rangle \langle id \rangle$   
 $\langle lowerId \rangle ::= \langle lower \rangle \langle id \rangle$   
 $\langle upper \rangle ::= \text{all upper case characters}$   
 $\langle lower \rangle ::= \text{all lower case characters}$   
 $\langle id \rangle ::= \text{a Haskell identifier}$ 
```

A Document Type Definition can contain comments using the rules Haskell uses for comments.

Haskell field labels (called *records* in the remainder) were added to the grammar for transparency with other tools. Because a $\langle field \rangle$ can have a labeled field, a $\langle prod \rangle$ definition can behave like a record statement. Therefore the parser can parse a record and fit it into the existing data type format, without any modifications to that type. This ensures that the back-end can stay the same as well. So adding records to the grammar and parser has a very low impact on the internal functionality.

The definition for $\langle prod \rangle$ changes to

```
 $\langle prod \rangle ::= \langle upperId \rangle (\langle field \rangle^* | \langle record \rangle) idps?$ 
```

and these rules are added:

```
 $\langle record \rangle ::= \{ \langle recField \rangle (, \langle recField \rangle)^* \}$   
 $\langle recField \rangle ::= \langle label \rangle (, \langle label \rangle)^* :: \langle type \rangle$ 
```


Appendix B

Installing Phi

To run an editor, we need the (modified) Proxima framework, the generic part of Phi, and of course Phi_{GHC} or Phi_{EHC} . The information in any README file may be more up-to-date than the information provided here.

B.1 Obtaining Proxima

Let $\$phi$ be the directory to install Phi. In $\$phi$, checkout the Phi branches of Proxima and the generator:

```
> svn checkout \  
> https://svn.cs.uu.nl:12443/repos/Proxima/proxima/branches/phi proxima  
  
> svn checkout \  
> https://svn.cs.uu.nl:12443/repos/Proxima/generator/branches/phi generator
```

Get the latest version of the UUAG Compiler and UULib (parsing and pretty printer library) from <http://www.cs.uu.nl/wiki/bin/view/HUT/Download>.

Check the paths of GHC and UUAG in the Makefile, found in $\$phi/proxima/src/Makefile$. Proxima uses Alex for the scanning process, which is needed as well.

B.2 Phi Common

Also in $\$phi$, checkout the generic part of Phi:

```
> svn checkout http://gerbo.servebeer.com/svn/phi-common
```

In the Makefile, $\$phi/phi-common/Makefile$, check the paths to the different programs. If you want to modify the source code, make sure that `cpphs` is installed, a C-macro preprocessor for Haskell.

The data type conversion tool makes use of **module** *Language.Haskell*, which can be found in package `haskell-src`.

B.3 Phi_{GHC}

To install Phi_{GHC} , checkout the editor in $\$phi$:

```
> svn checkout http://gerbo.servebeer.com/svn/phi-ghc
```

The Phi_{GHC} editor was tested with `ghc-6.8.2`; some modifications have to be made to the module that calls GHC and to the presentation sheet to make it compatible with `ghc-6.8.1` or `ghc-6.8.3`:

```
 $\$phi/phi-ghc/src/GhcParser.hs$   
 $\$phi/phi-ghc/src/PhiPresentationAG.ag$ 
```

Check the paths in the Makefile, `$phi/phi-ghc/Makefile`, although they should be correct.

In file `$phi/phi-ghc/src/PhiPresentationAG.ag` (or `$phi/phi-ghc/src/PresentationAG.hs`, if you do not want to rebuild a Haskell file from the UUAG file), check that `showToken` is defined to some dummy value (e.g. `showToken _ = ""`), unless you have a GHC version with debug information on (or, specifically, a GHC version that was built with **deriving** `Show` for `Lexer.Token`).

Building Phi

A *complete* rebuild requires the following steps: in `$phi/phi-ghc`, execute

```
> make generate
> make
```

These steps include rebuilding all generated files, including a Haskell file from a UUAG definition for the presentation sheet, which can take some time. If you only want to build `PhiGHC` without any modifications to the source code, the included pre-built and pre-generated files can be used, and only

```
> make nogeneration
```

is needed. This does not require `cpphs` to be installed, as the preprocessed files are included.

B.4 `PhiEHC`

The `PhiEHC` editor was tested with EHC version 100, commit 1182. EHC can be downloaded from <http://www.cs.uu.nl/wiki/bin/view/Ehc>. To add some debug information, we added

```
DERIVING *: Show
```

to

```
$EHC/src/ehc/HS/AbsSyn.cag
```

Alternatively, one can disable the debug message in `$phi/phi-common/ProxParser.hs`, on the line after the use of `compilerASTtoProxima`.

EHC was built with

```
> make 100/ehc
```

Checkout the editor in `$phi`:

```
> svn checkout http://gerbo.servebeer.com/svn/phi-ehc
```

Check the paths in the Makefile, `$phi/phi-ehc/Makefile`, although they should be correct.

To build `PhiEHC`, follow the steps as described under heading Building Phi, shown above (but in `$phi/phi-ehc` instead of `$phi/phi-ghc`).

Appendix C

Tutorial: Designing a Phi Editor

This appendix describes the Phi-specific things that are required to write a new or modify an existing editor based on Phi: first we set up the framework (Section C.1), then we describe how to use the combinators (Section C.2).

We assume the reader to know how to design an ordinary Proxima instantiation. This tutorial will help to understand the source code of Phi_{GHC} and Phi_{EHC} , but the source code of Phi_{GHC} and Phi_{EHC} may help to understand this tutorial as well. Names in this appendix may be different from the descriptions in Chapter 9.

C.1 Setting Up the Framework

First, we set up the framework: generate the Proxima DTD, access the external parser, and define the headers of different generated files to be able to use the types of the external compiler.

Document Type Generation

The module `Tailor.hs` is used by the tool that converts a set of Haskell data types to the Proxima Document Type Definition. The module contains a number of lists needed by the conversion tool to instruct the process of converting the parser data type definitions to the Proxima DTD:

<code>skipDataTypes :: [String]</code>	The parser data types that must be omitted. Used if the parser data type contains types that are not needed in the Proxima data type, or types that will be redefined in <i>extraDefs</i> or <i>specialMappings</i> (like treating a <i>Set</i> as a list: type <i>Set a</i> :: [<i>a</i>])
<code>skips :: [String]</code>	Mapping functions and type synonyms that must be omitted, such as <i>mapDocument</i> .
<code>skipsWithPat :: [(String, String)]</code>	In a mapping function, omit the arm with the given constructor pattern. Needed when that mapping is redefined in <i>specialMappings</i>
<code>extraDefs :: [String]</code>	Definitions that need to be added to the Proxima DTD, such as data <i>Document</i> = <i>RootDoc HsModule ExtraState</i> .
<code>specialMappings :: [String]</code>	Mapping functions for special cases. Mostly used to replace mappings in <i>skips</i> and <i>skipsWithPat</i> .
<code>wrapperTypes :: [(String, String)]</code>	Wrapper types with the unwrap functions. For example <code>[("Located", "unLoc")]</code> .
<code>argMapping :: [(String, [HsType])]</code>	Global type variable instantiations. <i>HsType</i> is from the <i>Language.Haskell.Syntax</i> library, in package <code>haskell-src</code> .
<code>specialArgMapping :: [(String, [(String, [HsType])])]</code>	Specific argument instantiations. Similar to the global argument instantiations, but it also specifies in what type to instantiate the arguments. Takes precedence over <i>argMapping</i> .
<code>passedArg :: String</code>	The name of an extra parameter passed through all generated mapping functions. Empty for no parameter.

Accessing the External Parser

The Haskell module `CompilerParser.hs` defines the root conversion mapping function, used in the generic Proxima parser.

```
compilerASTtoProxima = ⟨...⟩
```

and for that, it includes the generated conversion functions with the C-macro

```
#include "../gen/⟨...⟩.hs"
```

`CompilerParser.hs` also defines

```
parseString :: String → (Maybe mod, TokenStream, [CompilerParseError])
parseString = ⟨...⟩
type CompilerParseError = ⟨...⟩
type TokenStream = [⟨...⟩]
```

which actually calls the parser.

The module `Imports.hs` defines a number of imports, including

```
import ⟨...⟩ (CompilerParseError (.), TokenStream)
```

and all functions and types that are needed from the parser API in the presentation sheet. `Imports.hs` is included (`#include Imports.hs`) in `DocTypes_Generated.hs` and in the generic part of the presentation sheet. Any extra imports that are needed in the presentation sheet can be defined here.

Generated Files

Next we describe what needs to be added to the headers of the generated files.

Module `DocTypes_Generated.hs`

The header of `DocTypes_Generated.hs` gets the following extra definitions:

```
import Phi_Generated
#include "Imports.hs"
data UserToken = StringTk
                | OpTk
                deriving (Show, Eq, Ord, Data, Typeable)

data ExtraState = X { getTokStr      :: TokenStream
                    , getSourceCode  :: [String]
                    , getCheckedModule :: Maybe CheckedModule
                    , getParseErrors  :: [CompilerParseError]
                    }

instance Show ExtraState where
    show _ = "*ExtraState*"
instance Data ExtraState
instance Typeable ExtraState
```


Module DocumentEdit.Generated.ag

DocumentEdit.Generated needs some extra instances of *Editable*. Most are related to the types in *skipDataTypes* of Tailor.hs. Due to the change of the type of the parsing sheet, we need the following instance:

```
instance Editable EnrichedDoc Document Node ClipDoc UserToken
  => Editable (EnrichedDoc
    , WhitespaceMap
    , CommentMap Document Node ClipDoc UserToken
    , IDPCounter
    )
  Document Node ClipDoc UserToken
```

where

```
hole                = (hole, Map.empty, Map.empty, 1)
isList _            = False
insertList _ _ _    = Clip_Nothing
removeList _ _ _    = Clip_Nothing
select p (e, w, c, i) = select p e
paste p c (e, w, c', i) = (paste p c e, w, c', i)
alternatives (e, w, c, i) = alternatives e
arity (e, w, c, i)    = arity e
toClip (e, w, c, i)   = toClip e
fromClip c            = error "DocumentEdit.Generated: (,,) error"
parseErr e            = (parseErr e, Map.empty, Map.empty, 1)
holeNodeConstr _ _    = Node_HoleEnrichedDoc (error "bla") []
```

UUAG PresentationAG.Generated.ag

The header of PresentationAG.Generated.hs gets some macro's (to make it easier to make quick changes) and a function:

```
#define PRES \
  (loc.pres, loc.idps, (lhs.whitespaceMapCreated, lhs.commentMapCreated, _), lhs.tokStr, loc.noIdps)
#define __ADMINISTRATE_LIST (PRES, lhs.idps) = let x@(_, i, _, _) = __PROCESS in (x, i)
#define __ADMINISTRATE     PRES = __PROCESS
#define __PROCESS          process @lhs.tokStr @loc.pres' @idps @lhs.pIdC

process tokStr1 pres' idps1 pIdC
  = let (pres, idps2, wsMap, tokStr2) = pres' --
    in (pres, idps2, wsMap, tokStr2, length idps2 - length idps1)
```

The two administration macros are used in the generated body of the UUAG file.

Module Phi.Generated.hs

The module Phi.Generated.hs imports and exports most of the types present in the *skipDataTypes* list of Tailor.hs, and it defines a number of *Show*, *Data* and *Typeable* instances for those types. The generated converted types have to be included

```
#include "../gen/<...>ConvertedTypes.hs"
```

Furthermore, Phi.Generated.hs has to import all parser data types *qualified as Ext*, to let the generated mapping functions be able to refer to those types.

Template Presentation Sheet

The generator generates a template presentation sheet `PhiPresentationAG-template.ag`. For all data types and constructors, a default definition is generated, as well as the names of the available fields in comments for quick referencing. For example, for `PhiGHC` are generated

```
SEM HsExpr
| HsApp {-hsExpr1 | hsExpr2 -}
  loc.pres' = (empty, [], (@lhs.whitespaceMapCreated, @lhs.commentMapCreated
                        , @lhs.commentMap
                        ), @lhs.tokStr)
| OpApp {-hsExpr1 | hsExpr2 | fixity | hsExpr3 -}
  loc.pres' = (empty, [], (@lhs.whitespaceMapCreated, @lhs.commentMapCreated
                        , @lhs.commentMap
                        ), @lhs.tokStr)
```

Furthermore, default definitions for list-like structures are generated that administrate the IDP's. They can easily be instructed whether or not to use separator tokens between the list elements, but more on that later.

Presentation Sheet

First, an interface has to be implemented:

type <i>Token</i>	The type of a token in the token stream
type <i>ImplicitToken</i>	A representation of an optional token
type <i>IdName</i>	The type of identifiers in the parse tree
<i>isSameLoc</i> :: <i>Token</i> → <i>Token</i> → <i>Bool</i>	Determine whether two tokens have the same position in the source code
<i>tokElem</i> :: <i>Token</i> → [<i>ImplicitToken</i>] → <i>Bool</i>	Check whether a token is in a list of given implicit tokens
<i>token2String</i> :: <i>ImplicitToken</i> → <i>String</i>	A show for implicit tokens (not necessary for all tokens)
<i>showTokStr</i> :: <i>TokenStream</i> → <i>String</i>	Only needed for debugging; <i>showTokStr</i> _ = "" is sufficient
<i>startPos</i> :: <i>Token</i> → (<i>Int</i> , <i>Int</i>)	The line and column of the start position of a token (columns are 0-based)
<i>presName</i> :: [<i>IDP</i>] → <i>IDPCounter</i> → <i>Offset</i> → <i>IdName</i> → <i>WhitespaceMap</i> → <i>TokenStreamT</i> → <i>Bool</i> → <i>Pres'</i>	How to present an identifier. <i>TokenStreamT</i> is the token stream bundled with the source code. <i>Pres'</i> is the return type of (part of) <i>makePres</i>

Next, the semantic function for *EnrichedDoc* needs some basic setting up:

```
SEM EnrichedDoc
| RootEnr {-hsModule | extraState -}
  hsModule.tokStr = let tokStr = getTokStr @extraState
                    source = getSourceCode @extraState
                    startToken = ⟨...⟩ -- make a default token, with source position (0,0)
  in TokenStream
    ( error "first elt of tokStr, should not be evaluated"
    : startToken -- to handle starting whitespace/comments
    : tokStr
    ) source
  hsModule.whitespaceMapCreated = Map.empty
  hsModule.checkedModule = fromJust $ getCheckedModule @extraState
  hsModule.whitespaceMap = if Map.null @lhs.whitespaceMap
    then @hsModule.whitespaceMapCreated -- interpretation
    else @lhs.whitespaceMap           -- presentation
  hsModule.commentMap = @lhs.commentMap
```

C.2 Using the Combinators

The presentation sheet starts with a number of C-macro definitions that hide part of the overhead.

```
#define LAST_FLD(child)      @idps @lhs.pIdC _WT(child)
#define OTHER(child)        Other @child.pres @child.tokStr
#define LIST(child)         List @child.pres @child.tokStr @child.idps @child.offset
#define LIST_ADM(child, n, sep) (child.offset, child.sepSigns) = (n, sep)
#define _WT(attr)           (@attr.whitespaceMapCreated, @attr.commentMapCreated \
                             , @lhs.commentMap) @lhs.tokStr @lhs.isInterp
```

First is the `LAST_FLD` macro, which can be seen as the second argument to `makePres`, as it is always used as

```
loc.phiPres = makePres [⟨...⟩] LAST_FLD(⟨...⟩)
```

The argument of `LAST_FLD` is the name of the last child of the node, or `lhs` if it is a leaf-node.

The `OTHER` macro is used for the `Other` presentation command, and threads the presentation and token stream of its argument into `makePres`.

`LIST` is used if a child node is a list-like structure in which the elements have to be separated by tokens. Whenever a `LIST` is used, `LIST_ADM` has to be used too. The parameters of `LIST_ADM` are the name of the child node that is the list-like structure, the offset, and the list separator token.

`_WT` is used in `LAST_FLD`.

We will look at a simple and a complex presentation, both taken out of the presentation sheet of `PhiGHC`, used to present a let expression (Listing C.1) and a class declaration (Listing C.2). They use different alternatives of `makePres`: `makePres2` and `makePres4'`. The appended number corresponds to the size of the returned tuple minus one; every time an `OTHER` command follows on a `Token`, `Skip`, `OptionalToken`, or `PresName` command, the number must be increased. The presence of a prime indicates that the offsets for the token streams are returned as well.

`PresName` is an extra presentation command to make it easier to present the often occurring identifiers.

The attribute `loc.phiPres` is a tuple with the presentation, created IDP's, the whitespace map that is being built, and the token stream for `lhs`.

The example in Listing C.1 is straightforward; two tokens with the presentation for the bindings in between, and the presentation of the expression at the end. `HsLocalBinds` is a separate constructor that contains the list of bindings and displays the optional curly brackets. Listing C.2 is more complex: it presents three lists of which the elements may be separated by tokens. The type variables are also in a list structure, but the elements are *not* separated by tokens and therefore it can be presented using the `OTHER` macro.

```
SEM HsExpr
<...>
| HsLet {-hsLocalBinds | hsExpr -}
  (loc.phiPres, hsLocalBinds.tokStr, hsExpr.tokStr)
  = makePres2 [Token $ key "let"
               , OTHER(hsLocalBinds)
               , Token $ key "in"
               , OTHER(hsExpr)
               ] LAST_FLD(hsExpr)
```

Listing C.1: Fragment of the presentation sheet of Phi_{GHC}: the simple **let** expression

```
SEM TypeClassDecl
<...>
| ClassDecl {-context | name | typeVars | funDeps | mixed -}
  (loc.phiPres
  , (context.tokStr , loc.n1)
  , (typeVars.tokStr, _ )
  , (funDeps.tokStr , loc.n2)
  , (mixed.tokStr   , loc.n3)
  ) = makePres4' [Token (key "class")
                 -- class context
                 , LIST(context)
                 , if null @context.press
                   then Skip
                   else Token $ sym "=>"

                 -- class name + type variables
                 , OptionalToken openParenTok
                 , PresName @name -- name
                 , OTHER(typeVars) -- tyvars
                 , OptionalToken closeParenTok

                 -- functional dependencies
                 , if null @funDeps.press
                   then Skip
                   else Token $ sym "|"
                 , LIST(funDeps)

                 -- check whether the 'where' has a body
                 , if null @mixed.press
                   then Skip
                   else Token $ sym "where"

                 -- the body of the where
                 , OptionalToken openCurly
                 , LIST(mixed)
                 , OptionalToken closeCurly
                 ] LAST_FLD(mixed)
  LIST_ADM(context, @loc.n1, comma)
  LIST_ADM(funDeps, @loc.n2, comma)
  LIST_ADM(mixed, @loc.n3, semi)
```

Listing C.2: Fragment of the presentation sheet of Phi_{GHC}: the complex **class** declaration

Bibliography

- [AM05] Krasimir Angelov and Simon Marlow. Visual Haskell: a full-featured haskell development environment. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 5–16, New York, NY, USA, 2005. ACM.
- [BEK⁺05] Johan Broberg, Tobias Engvall, Lennart Kolmodin, Rickard Nilsson, and David Waern. Haste – Haskell TurboEdit. Technical report, Chalmers University of Technology, 2005.
- [Dij] Atze Dijkstra. EHC Web. <http://www.cs.uu.nl/wiki/Ehc/WebHome>.
- [FLMJ98] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: a binary foreign language interface for haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 153–162, New York, NY, USA, 1998. ACM.
- [Fre07] Leif Frenzel. Experience report: building an Eclipse-based IDE for Haskell. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 220–222, New York, NY, USA, 2007. ACM.
- [GHC] GHC Team. The Glasgow Haskell Compiler. <http://www.haskell.org/ghc/>.
- [Jon05] I. Jones. The Haskell Cabal, a common architecture for building applications and libraries. In *6th Symposium on Trends in Functional Programming*, pages 340–354, 2005. <http://www.haskell.org/cabal/>.
- [Lei04] Daan Leijen. wxHaskell: a portable and concise GUI library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM.
- [LTR] Huiqing Li, Simon Thompson, and Claus Reinke. HaRe – The Haskell Refactorer. <http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html>.
- [LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of LNCS, pages 357–375. Springer-Verlag, January 2003.
- [Mar02] Simon Marlow. Haddock, a Haskell documentation tool. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 78–89, New York, NY, USA, 2002. ACM.
- [NF08] Jürgen Nicklisch-Franken. *Leksah: An Integrated Development Environment for Haskell*. <http://leksah.org/>, 2008.
- [Pey03] S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.
- [RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984.
- [RT05] Chris Ryder and Simon Thompson. Porting HaRe to the GHC API. Technical Report 8-05, Computing Laboratory, University of Kent, Canterbury, Kent, UK, October 2005.

- [SB04] S. Doaitse Swierstra and Arthur Baars. *Attribute Grammar System*. Universiteit Utrecht, The Netherlands, <http://www.cs.uu.nl/groups/ST/Center/AttributeGrammarSystem>, 2004.
- [SC05] Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, September 2005.
- [Sch04] Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Universiteit Utrecht, The Netherlands, Oct 2004.
- [SJ01] Martijn M. Schrage and Johan Jeuring. Xprez: A declarative presentation language for xml, 2001.
- [SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM.
- [Sve02] Jonas Svensson. *hIDE User manual*, 2002. <http://www.dtek.chalmers.se/~d99josve/hide/>.
- [Swi08] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.